TikZ & PGF

Manual for Version 3.1.12

```
\begin{tikzpicture}
  \coordinate (front) at (0,0);
  \coordinate (horizon) at (0,.31\paperheight);
  \coordinate (bottom) at (0,-.6\paperheight);
  \coordinate (sky) at (0,.57\paperheight);
  \coordinate (left) at (-.51\paperwidth,0);
  \coordinate (right) at (.51\paperwidth,0);

  \shade [bottom color=white,
          top color=blue!30!black!50]
          ([yshift=-5mm]horizon -|  left)
    rectangle (sky -| right);

  \shade [bottom color=black!70!green!25,
          top color=black!70!green!10]
    (front -| left) -- (horizon -| left)
    decorate [decoration=random steps] {
      -- (horizon -| right)  }
    -- (front -| right) -- cycle;

  \shade [top color=black!70!green!25,
        bottom color=black!25]
              ([yshift=-5mm-1pt]front -| left)
    rectangle ([yshift=1pt]front -| right);

  \fill [black!25]
              (bottom -| left)
    rectangle ([yshift=-5mm]front -| right);

  \def\nodeshadowed[#1]#2;{
    \node[scale=2,above,#1]{
      \global\setbox\mybox=\hbox{#2}
      \copy\mybox};
    \node[scale=2,above,#1,yscale=-1,
          scope fading=south,opacity=0.4]{\box\mybox};
  }

  \nodeshadowed [at={(-5,8  )},yslant=0.05]
    {\Huge Ti\textcolor{orange}{\emph{k}}Z};
  \nodeshadowed [at={( 0,8.3)}]
    {\huge \textcolor{green!50!black!50}{\&}};
  \nodeshadowed [at={( 5,8  )},yslant=-0.05]
    {\Huge \textsc{PGF}};
  \nodeshadowed [at={( 0,5  )}]
    {Manual for Version \pgftypesetversion};

  \foreach \where in {-9cm,9cm} {
    \nodeshadowed [at={(\where,5cm)}] { \tikz
      \draw [green!20!black, rotate=90,
             l-system={rule set={F -> FF-[-F+F]+[+F-F]},
               axiom=F, order=4,step=2pt,
               randomize step percent=50, angle=30,
               randomize angle percent=5}] l-system; }}

  \foreach \i in {0.5,0.6,...,2}
    \fill
      [white,opacity=\i/2,
       decoration=Koch snowflake,
       shift=(horizon),shift={(rand*11,rnd*7)},
       scale=\i,double copy shadow={
         opacity=0.2,shadow xshift=0pt,
         shadow yshift=3*\i pt,fill=white,draw=none}]
      decorate {
        decorate {
          decorate {
            (0,0)- ++(60:1) -- ++(-60:1) -- cycle
          } } };

  \node (left text) ...
  \node (right text) ...

  \fill [decorate,decoration={footprints,foot of=gnome},
         opacity=.5,brown]         (rand*8,-rnd*10)
    to [out=rand*180,in=rand*180] (rand*8,-rnd*10);
\end{tikzpicture}
```

Für meinen Vater, damit er noch viele schöne TEX-Graphiken erschaffen kann.

Till

Copyright 2007 to 2013 by Till Tantau

Permission is granted to copy, distribute and/or modify *the documentation* under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

Permission is granted to copy, distribute and/or modify *the code of the package* under the terms of the GNU Public License, Version 2 or any later version published by the Free Software Foundation. A copy of the license is included in the section entitled GNU Public License.

Permission is also granted to distribute and/or modify *both the documentation and the code* under the conditions of the LaTeX Project Public License, either version 1.3 of this license or (at your option) any later version. A copy of the license is included in the section entitled LATEX Project Public License.

The Ti*k*Z and PGF Packages

Manual for version 3.1.12

https://github.com/pgf-tikz/pgf

Till Tantau*
Institut für Theoretische Informatik
Universität zu Lübeck

November 6, 2026

Contents

1 Introduction **27**
1.1 The Layers Below Ti*k*Z . . . 27
1.2 Comparison with Other Graphics Packages . . . 28
1.3 Utility Packages . . . 28
1.4 How to Read This Manual . . . 29
1.5 Authors and Acknowledgements . . . 29
1.6 Getting Help . . . 29

I Tutorials and Guidelines **30**

2 Tutorial: A Picture for Karl's Students **31**
2.1 Problem Statement . . . 31
2.2 Setting up the Environment . . . 31
2.2.1 Setting up the Environment in LaTeX . . . 31
2.2.2 Setting up the Environment in Plain TeX . . . 32
2.2.3 Setting up the Environment in ConTeXt . . . 32
2.3 Straight Path Construction . . . 33
2.4 Curved Path Construction . . . 33
2.5 Circle Path Construction . . . 34
2.6 Rectangle Path Construction . . . 34
2.7 Grid Path Construction . . . 35
2.8 Adding a Touch of Style . . . 35
2.9 Drawing Options . . . 36
2.10 Arc Path Construction . . . 36
2.11 Clipping a Path . . . 37
2.12 Parabola and Sine Path Construction . . . 38
2.13 Filling and Drawing . . . 38
2.14 Shading . . . 39
2.15 Specifying Coordinates . . . 40
2.16 Intersecting Paths . . . 41
2.17 Adding Arrow Tips . . . 41
2.18 Scoping . . . 42
2.19 Transformations . . . 43
2.20 Repeating Things: For-Loops . . . 43
2.21 Adding Text . . . 45
2.22 Pics: The Angle Revisited . . . 48

*Editor of this documentation. Parts of this documentation have been written by other authors as indicated in these parts or chapters and in Section 1.5.

3 Tutorial: A Petri-Net for Hagen **50**
3.1 Problem Statement . . . 50
3.2 Setting up the Environment . . . 50
3.2.1 Setting up the Environment in LaTeX . . . 50
3.2.2 Setting up the Environment in Plain TeX . . . 50
3.2.3 Setting up the Environment in ConTeXt . . . 51
3.3 Introduction to Nodes . . . 51
3.4 Placing Nodes Using the At Syntax . . . 52
3.5 Using Styles . . . 52
3.6 Node Size . . . 53
3.7 Naming Nodes . . . 53
3.8 Placing Nodes Using Relative Placement . . . 54
3.9 Adding Labels Next to Nodes . . . 54
3.10 Connecting Nodes . . . 56
3.11 Adding Labels Next to Lines . . . 58
3.12 Adding the Snaked Line and Multi-Line Text . . . 58
3.13 Using Layers: The Background Rectangles . . . 59
3.14 The Complete Code . . . 60

4 Tutorial: Euclid's Amber Version of the *Elements* **62**
4.1 Book I, Proposition I . . . 62
4.1.1 Setting up the Environment . . . 62
4.1.2 The Line AB . . . 63
4.1.3 The Circle Around A . . . 63
4.1.4 The Intersection of the Circles . . . 65
4.1.5 The Complete Code . . . 66
4.2 Book I, Proposition II . . . 67
4.2.1 Using Partway Calculations for the Construction of D . . . 67
4.2.2 Intersecting a Line and a Circle . . . 68
4.2.3 The Complete Code . . . 69

5 Tutorial: Diagrams as Simple Graphs **70**
5.1 Styling the Nodes . . . 70
5.2 Aligning the Nodes Using Positioning Options . . . 72
5.3 Aligning the Nodes Using Matrices . . . 74
5.4 The Diagram as a Graph . . . 75
5.4.1 Connecting Already Positioned Nodes . . . 75
5.4.2 Creating Nodes Using the Graph Command . . . 76

6 Tutorial: A Lecture Map for Johannes **80**
6.1 Problem Statement . . . 80
6.2 Introduction to Trees . . . 80
6.3 Creating the Lecture Map . . . 83
6.4 Adding the Lecture Annotations . . . 87
6.5 Adding the Background . . . 88
6.6 Adding the Calendar . . . 89
6.7 The Complete Code . . . 91

7 Guidelines on Graphics **95**
7.1 Planning the Time Needed for the Creation of Graphics . . . 95
7.2 Workflow for Creating a Graphic . . . 95
7.3 Linking Graphics With the Main Text . . . 96
7.4 Consistency Between Graphics and Text . . . 96
7.5 Labels in Graphics . . . 97
7.6 Plots and Charts . . . 97
7.7 Attention and Distraction . . . 100

II Installation and Configuration **102**

8 Installation **103**
8.1 Package and Driver Versions . . . 103
8.2 Installing Prebundled Packages . . . 103
8.2.1 Debian . . . 103
8.2.2 MiKTeX . . . 103
8.3 Installation in a texmf Tree . . . 104
8.3.1 Installation that Keeps Everything Together . . . 104
8.3.2 Installation that is TDS-Compliant . . . 104
8.4 Updating the Installation . . . 104

9 Licenses and Copyright **105**
9.1 Which License Applies? . . . 105
9.2 The GNU Public License, Version 2 . . . 105
9.2.1 Preamble . . . 105
9.2.2 Terms and Conditions For Copying, Distribution and Modification . . . 106
9.2.3 No Warranty . . . 108
9.3 The LaTeX Project Public License, Version 1.3c 2006-05-20 . . . 108
9.3.1 Preamble . . . 108
9.3.2 Definitions . . . 108
9.3.3 Conditions on Distribution and Modification . . . 109
9.3.4 No Warranty . . . 110
9.3.5 Maintenance of The Work . . . 111
9.3.6 Whether and How to Distribute Works under This License . . . 111
9.3.7 Choosing This License or Another License . . . 111
9.3.8 A Recommendation on Modification Without Distribution . . . 112
9.3.9 How to Use This License . . . 112
9.3.10 Derived Works That Are Not Replacements . . . 112
9.3.11 Important Recommendations . . . 112
9.4 GNU Free Documentation License, Version 1.2, November 2002 . . . 113
9.4.1 Preamble . . . 113
9.4.2 Applicability and definitions . . . 113
9.4.3 Verbatim Copying . . . 114
9.4.4 Copying in Quantity . . . 114
9.4.5 Modifications . . . 114
9.4.6 Combining Documents . . . 116
9.4.7 Collection of Documents . . . 116
9.4.8 Aggregating with independent Works . . . 116
9.4.9 Translation . . . 116
9.4.10 Termination . . . 116
9.4.11 Future Revisions of this License . . . 117
9.4.12 Addendum: How to use this License for your documents . . . 117

10 Supported Formats **118**
10.1 Supported Input Formats: LaTeX, Plain TeX, ConTeXt . . . 118
10.1.1 Using the LaTeX Format . . . 118
10.1.2 Using the Plain TeX Format . . . 118
10.1.3 Using the ConTeXt Format . . . 118
10.2 Supported Output Formats . . . 119
10.2.1 Selecting the Backend Driver . . . 119
10.2.2 Producing PDF Output . . . 119
10.2.3 Producing PostScript Output . . . 120
10.2.4 Producing SVG Output . . . 120
10.2.5 Producing Perfectly Portable DVI Output . . . 122

III Ti*k*Z ist *kein* Zeichenprogramm **123**

11 Design Principles **124**
11.1 Special Syntax For Specifying Points . . . 124
11.2 Special Syntax For Path Specifications . . . 124
11.3 Actions on Paths . . . 125
11.4 Key–Value Syntax for Graphic Parameters . . . 125
11.5 Special Syntax for Specifying Nodes . . . 125
11.6 Special Syntax for Specifying Trees . . . 125
11.7 Special Syntax for Graphs . . . 126
11.8 Grouping of Graphic Parameters . . . 126
11.9 Coordinate Transformation System . . . 127

12 Hierarchical Structures: Package, Environments, Scopes, and Styles **128**
12.1 Loading the Package and the Libraries . . . 128
12.2 Creating a Picture . . . 128
12.2.1 Creating a Picture Using an Environment . . . 128
12.2.2 Creating a Picture Using a Command . . . 130
12.2.3 Handling Catcodes and the Babel Package . . . 130
12.2.4 Adding a Background . . . 131
12.3 Using Scopes to Structure a Picture . . . 131
12.3.1 The Scope Environment . . . 131
12.3.2 Shorthand for Scope Environments . . . 132
12.3.3 Single Command Scopes . . . 132
12.3.4 Using Scopes Inside Paths . . . 133
12.4 Using Graphic Options . . . 133
12.4.1 How Graphic Options Are Processed . . . 133
12.4.2 Using Styles to Manage How Pictures Look . . . 133

13 Specifying Coordinates **135**
13.1 Overview . . . 135
13.2 Coordinate Systems . . . 135
13.2.1 Canvas, XYZ, and Polar Coordinate Systems . . . 135
13.2.2 Barycentric Systems . . . 138
13.2.3 Node Coordinate System . . . 139
13.2.4 Tangent Coordinate Systems . . . 141
13.2.5 Defining New Coordinate Systems . . . 142
13.3 Coordinates at Intersections . . . 142
13.3.1 Intersections of Perpendicular Lines . . . 143
13.3.2 Intersections of Arbitrary Paths . . . 143
13.4 Relative and Incremental Coordinates . . . 145
13.4.1 Specifying Relative Coordinates . . . 145
13.4.2 Rotational Relative Coordinates . . . 146
13.4.3 Relative Coordinates and Scopes . . . 147
13.5 Coordinate Calculations . . . 147
13.5.1 The General Syntax . . . 148
13.5.2 The Syntax of Factors . . . 148
13.5.3 The Syntax of Partway Modifiers . . . 149
13.5.4 The Syntax of Distance Modifiers . . . 150
13.5.5 The Syntax of Projection Modifiers . . . 150

14 Syntax for Path Specifications **152**
14.1 The Move-To Operation . . . 154
14.2 The Line-To Operation . . . 154
14.2.1 Straight Lines . . . 154
14.2.2 Horizontal and Vertical Lines . . . 155
14.3 The Curve-To Operation . . . 155
14.4 The Rectangle Operation . . . 156
14.5 Rounding Corners . . . 156
14.6 The Circle and Ellipse Operations . . . 157
14.7 The Arc Operation . . . 158

14.8 The Grid Operation . . . 159
14.9 The Parabola Operation . . . 161
14.10 The Sine and Cosine Operation . . . 162
14.11 The SVG Operation . . . 163
14.12 The Plot Operation . . . 163
14.13 The To Path Operation . . . 163
14.14 The Foreach Operation . . . 166
14.15 The Let Operation . . . 167
14.16 The Scoping Operation . . . 168
14.17 The Node and Edge Operations . . . 168
14.18 The Graph Operation . . . 169
14.19 The Pic Operation . . . 169
14.20 The Attribute Animation Operation . . . 169
14.21 The PGF-Extra Operation . . . 169
14.22 Interacting with the Soft Path subsystem . . . 170

15 Actions on Paths **171**
15.1 Overview . . . 171
15.2 Specifying a Color . . . 172
15.3 Drawing a Path . . . 172
15.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join . . . 173
15.3.2 Graphic Parameters: Dash Pattern . . . 174
15.3.3 Graphic Parameters: Draw Opacity . . . 176
15.3.4 Graphic Parameters: Double Lines and Bordered Lines . . . 177
15.4 Adding Arrow Tips to a Path . . . 178
15.5 Filling a Path . . . 178
15.5.1 Graphic Parameters: Fill Pattern . . . 179
15.5.2 Graphic Parameters: Interior Rules . . . 180
15.5.3 Graphic Parameters: Fill Opacity . . . 180
15.6 Generalized Filling: Using Arbitrary Pictures to Fill a Path . . . 180
15.7 Shading a Path . . . 182
15.8 Establishing a Bounding Box . . . 183
15.9 Clipping and Fading (Soft Clipping) . . . 185
15.10 Doing Multiple Actions on a Path . . . 186
15.11 Decorating and Morphing a Path . . . 188

16 Arrows **189**
16.1 Overview . . . 189
16.2 Where and When Arrow Tips Are Placed . . . 189
16.3 Arrow Keys: Configuring the Appearance of a Single Arrow Tip . . . 191
16.3.1 Size . . . 192
16.3.2 Scaling . . . 195
16.3.3 Arc Angles . . . 195
16.3.4 Slanting . . . 196
16.3.5 Reversing, Halving, Swapping . . . 196
16.3.6 Coloring . . . 197
16.3.7 Line Styling . . . 199
16.3.8 Bending and Flexing . . . 201
16.4 Arrow Tip Specifications . . . 205
16.4.1 Syntax . . . 205
16.4.2 Specifying Paddings . . . 206
16.4.3 Specifying the Line End . . . 207
16.4.4 Defining Shorthands . . . 207
16.4.5 Scoping of Arrow Keys . . . 209
16.5 Reference: Arrow Tips . . . 210
16.5.1 Barbed Arrow Tips . . . 212
16.5.2 Mathematical Barbed Arrow Tips . . . 215
16.5.3 Geometric Arrow Tips . . . 216
16.5.4 Caps . . . 219

16.5.5 Special Arrow Tips . . . 221

17 Nodes and Edges **222**
17.1 Overview . . . 222
17.2 Nodes and Their Shapes . . . 222
17.2.1 Syntax of the Node Command . . . 222
17.2.2 Predefined Shapes . . . 227
17.2.3 Common Options: Separations, Margins, Padding and Border Rotation . . . 227
17.3 Multi-Part Nodes . . . 230
17.4 The Node Text . . . 231
17.4.1 Text Parameters: Color and Opacity . . . 231
17.4.2 Text Parameters: Font . . . 231
17.4.3 Text Parameters: Alignment and Width for Multi-Line Text . . . 232
17.4.4 Text Parameters: Height and Depth of Text . . . 236
17.5 Positioning Nodes . . . 236
17.5.1 Positioning Nodes Using Anchors . . . 236
17.5.2 Basic Placement Options . . . 237
17.5.3 Advanced Placement Options . . . 238
17.5.4 Advanced Arrangements of Nodes . . . 242
17.6 Fitting Nodes to a Set of Coordinates . . . 243
17.7 Transformations . . . 243
17.8 Placing Nodes on a Line or Curve Explicitly . . . 244
17.9 Placing Nodes on a Line or Curve Implicitly . . . 248
17.10 The Label and Pin Options . . . 248
17.10.1 Overview . . . 248
17.10.2 The Label Option . . . 249
17.10.3 The Pin Option . . . 251
17.10.4 The Quotes Syntax . . . 252
17.11 Connecting Nodes: Using Nodes as Coordinates . . . 255
17.12 Connecting Nodes: Using the Edge Operation . . . 256
17.12.1 Basic Syntax of the Edge Operation . . . 256
17.12.2 Nodes on Edges: Quotes Syntax . . . 257
17.13 Referencing Nodes Outside the Current Picture . . . 258
17.13.1 Referencing a Node in a Different Picture . . . 258
17.13.2 Referencing the Current Page Node – Absolute Positioning . . . 259
17.14 Late Code and Late Options . . . 259

18 Pics: Small Pictures on Paths **261**
18.1 Overview . . . 261
18.2 The Pic Syntax . . . 261
18.2.1 The Quotes Syntax . . . 265
18.3 Defining New Pic Types . . . 266

19 Specifying Graphs **267**
19.1 Overview . . . 267
19.2 Concepts . . . 268
19.2.1 Concept: Node Chains . . . 268
19.2.2 Concept: Chain Groups . . . 269
19.2.3 Concept: Edge Labels and Styles . . . 270
19.2.4 Concept: Node Sets . . . 271
19.2.5 Concept: Graph Macros . . . 271
19.2.6 Concept: Graph Expressions and Color Classes . . . 272
19.3 Syntax of the Graph Path Command . . . 272
19.3.1 The Graph Command . . . 272
19.3.2 Syntax of Group Specifications . . . 274
19.3.3 Syntax of Chain Specifications . . . 276
19.3.4 Syntax of Node Specifications . . . 277
19.3.5 Specifying Tries . . . 283
19.4 Quick Graphs . . . 284

19.5 Simple Versus Multi-Graphs . . . 285
19.6 Graph Edges: Labeling and Styling . . . 287
19.6.1 Options For All Edges Between Two Groups . . . 287
19.6.2 Changing Options For Certain Edges . . . 288
19.6.3 Options For Incoming and Outgoing Edges . . . 288
19.6.4 Special Syntax for Options For Incoming and Outgoing Edges . . . 290
19.6.5 Placing Node Texts on Incoming Edges . . . 291
19.7 Graph Operators, Color Classes, and Graph Expressions . . . 291
19.7.1 Color Classes . . . 292
19.7.2 Graph Operators on Groups of Nodes . . . 293
19.7.3 Graph Operators for Joining Groups . . . 296
19.8 Graph Macros . . . 297
19.9 Online Placement Strategies . . . 299
19.9.1 Manual Placement . . . 299
19.9.2 Placement on a Grid . . . 300
19.9.3 Placement Taking Node Sizes Into Account . . . 303
19.9.4 Placement On a Circle . . . 305
19.9.5 Levels and Level Styles . . . 307
19.9.6 Defining New Online Placement Strategies . . . 308
19.10 Reference: Predefined Elements . . . 311
19.10.1 Graph Macros . . . 311
19.10.2 Group Operators . . . 314
19.10.3 Joining Operators . . . 315

20 Matrices and Alignment 318
20.1 Overview . . . 318
20.2 Matrices are Nodes . . . 318
20.3 Cell Pictures . . . 319
20.3.1 Alignment of Cell Pictures . . . 319
20.3.2 Setting and Adjusting Column and Row Spacing . . . 320
20.3.3 Cell Styles and Options . . . 322
20.4 Anchoring a Matrix . . . 326
20.5 Considerations Concerning Active Characters . . . 326
20.6 Examples . . . 327

21 Making Trees Grow 331
21.1 Introduction to the Child Operation . . . 331
21.2 Child Paths and Child Nodes . . . 332
21.3 Naming Child Nodes . . . 332
21.4 Specifying Options for Trees and Children . . . 333
21.5 Placing Child Nodes . . . 334
21.5.1 Basic Idea . . . 334
21.5.2 Default Growth Function . . . 335
21.5.3 Missing Children . . . 337
21.5.4 Custom Growth Functions . . . 338
21.6 Edges From the Parent Node . . . 338

22 Plots of Functions 341
22.1 Overview . . . 341
22.2 The Plot Path Operation . . . 341
22.3 Plotting Points Given Inline . . . 342
22.4 Plotting Points Read From an External File . . . 342
22.5 Plotting a Function . . . 343
22.6 Plotting a Function Using Gnuplot . . . 344
22.7 Placing Marks on the Plot . . . 346
22.8 Smooth Plots, Sharp Plots, Jump Plots, Comb Plots and Bar Plots . . . 348

23 Transparency **353**
23.1 Overview . . . 353
23.2 Specifying a Uniform Opacity . . . 353
23.3 Blend Modes . . . 355
23.4 Fadings . . . 358
23.4.1 Creating Fadings . . . 358
23.4.2 Fading a Path . . . 360
23.4.3 Fading a Scope . . . 362
23.5 Transparency Groups . . . 363

24 Decorated Paths **365**
24.1 Overview . . . 365
24.2 Decorating a Subpath Using the Decorate Path Command . . . 367
24.3 Decorating a Complete Path . . . 369
24.4 Adjusting Decorations . . . 370
24.4.1 Positioning Decorations Relative to the To-Be-Decorate Path . . . 370
24.4.2 Starting and Ending Decorations Early or Late . . . 371

25 Transformations **373**
25.1 The Different Coordinate Systems . . . 373
25.2 The XY- and XYZ-Coordinate Systems . . . 373
25.3 Coordinate Transformations . . . 374
25.4 Canvas Transformations . . . 378

26 Animations **380**
26.1 Introduction . . . 380
26.1.1 Animations Change Attributes . . . 381
26.1.2 Limitations of the Animation System . . . 381
26.1.3 Concepts: (Graphic) Objects . . . 381
26.1.4 Concepts: Attributes . . . 382
26.1.5 Concepts: Timelines . . . 382
26.2 Creating an Animation . . . 382
26.2.1 The Animate Key . . . 382
26.2.2 Timeline Entries . . . 383
26.2.3 Specifying Objects . . . 384
26.2.4 Specifying Attributes . . . 385
26.2.5 Specifying IDs . . . 385
26.2.6 Specifying Times . . . 386
26.2.7 Values . . . 388
26.2.8 Scopes . . . 388
26.3 Syntactic Simplifications . . . 389
26.3.1 The Colon Syntax I: Specifying Objects and Attributes . . . 389
26.3.2 The Colon Syntax II: Animating Myself . . . 390
26.3.3 The Time Syntax: Specifying Times . . . 391
26.3.4 The Quote Syntax: Specifying Values . . . 392
26.3.5 Timesheets . . . 392
26.4 The Attributes That Can Be Animated . . . 393
26.4.1 Animating Color, Opacity, and Visibility . . . 394
26.4.2 Animating Paths and their Rendering . . . 396
26.4.3 Animating Transformations: Relative Transformations . . . 398
26.4.4 Animating Transformations: Positioning . . . 402
26.4.5 Animating Transformations: Views . . . 403
26.5 Controlling the Timeline . . . 404
26.5.1 Before and After the Timeline: Value Filling . . . 404
26.5.2 Beginning and Ending Timelines . . . 405
26.5.3 Repeating Timelines and Accumulation . . . 409
26.5.4 Smoothing and Jumping Timelines . . . 410
26.6 Snapshots . . . 412

IV Graph Drawing **415**

27 Introduction to Algorithmic Graph Drawing **416**
27.1 What Is Algorithmic Graph Drawing? . . . 416
27.2 Using the Graph Drawing System . . . 416
27.3 Extending the Graph Drawing System . . . 417
27.4 The Layers of the Graph Drawing System . . . 418
27.5 Organisation of the Graph Drawing Documentation . . . 418
27.6 Acknowledgements . . . 419

28 Using Graph Drawing in Ti*k*Z **420**
28.1 Choosing a Layout and a Library . . . 420
28.2 Graph Drawing Parameters . . . 421
28.3 Padding and Node Distances . . . 422
28.4 Anchoring a Graph . . . 425
28.5 Orienting a Graph . . . 428
28.6 Fine-Tuning Positions of Nodes . . . 432
28.7 Packing of Connected Components . . . 433
28.7.1 Ordering the Components . . . 434
28.7.2 Arranging Components in a Certain Direction . . . 435
28.7.3 Aligning Components . . . 436
28.7.4 The Distance Between Components . . . 441
28.8 Anchoring Edges . . . 441
28.9 Hyperedges . . . 443
28.10 Using Several Different Layouts to Draw a Single Graph . . . 443
28.10.1 Sublayouts . . . 443
28.10.2 Subgraph Nodes . . . 445
28.10.3 Overlapping Sublayouts . . . 447
28.11 Miscellaneous Options . . . 448

29 Using Graph Drawing in PGF **450**
29.1 Overview . . . 450
29.2 How Graph Drawing in PGF Works . . . 450
29.2.1 Graph Drawing Scopes . . . 451
29.3 Layout Scopes . . . 454
29.4 Layout Keys . . . 454
29.5 Parameters . . . 456
29.6 Events . . . 456
29.7 Subgraph Nodes . . . 457

30 Graph Drawing Layouts: Trees **458**
30.1 The Tree Layouts . . . 458
30.1.1 The Reingold–Tilford Layout . . . 458
30.2 Specifying Missing Children . . . 463
30.3 Spanning Tree Computation . . . 465

31 Graph Drawing Algorithms: Layered Layouts **468**
31.1 The Modular Sugiyama Method . . . 468
31.2 Cycle Removal . . . 470
31.3 Layer Assignment (Node Ranking) . . . 471
31.4 Crossing Minimization (Node Ordering) . . . 471
31.5 Node Positioning (Coordinate Assignment) . . . 472
31.6 Edge Routing . . . 472

32 Graph Drawing Algorithms: Force-Based Methods 473
32.1 Controlling and Configuring Force-Based Algorithms . . . 474
32.1.1 Start Configuration . . . 474
32.1.2 The Iterative Process and Cooling . . . 474
32.1.3 Forces and Their Effects: Springs . . . 476
32.1.4 Forces and Their Effects: Electrical Repulsion . . . 477
32.1.5 Coarsening . . . 478
32.2 Spring Layouts . . . 479
32.3 Spring Electrical Layouts . . . 479

33 Graph Drawing Algorithms: Circular Layouts 481

34 Graph Drawing Layouts: Phylogenetic Trees 483
34.1 Generating a Phylogenetic Tree . . . 483
34.2 Laying out the Phylogram . . . 485

35 Graph Drawing Algorithms: Edge Routing 488

36 The Algorithm Layer 489
36.1 Overview . . . 489
36.2 Getting Started . . . 490
36.2.1 The Hello World of Graph Drawing . . . 490
36.2.2 Declaring an Algorithm . . . 491
36.2.3 The Run Method . . . 491
36.2.4 Loading Algorithms on Demand . . . 492
36.2.5 Declaring Options . . . 492
36.2.6 Adding Inline Documentation . . . 493
36.2.7 Adding External Documentation . . . 493
36.3 Namespaces and File Names . . . 495
36.3.1 Namespaces . . . 495
36.3.2 Defining and Using Namespaces and Classes . . . 496
36.4 The Graph Drawing Scope . . . 497
36.5 The Model Classes . . . 498
36.5.1 Directed Graphs (Digraphs) . . . 498
36.5.2 Vertices . . . 503
36.5.3 Arcs . . . 506
36.5.4 Edges . . . 510
36.5.5 Collections . . . 511
36.5.6 Coordinates, Paths, and Transformations . . . 513
36.5.7 Options and Data Storages for Vertices, Arcs, and Digraphs . . . 519
36.5.8 Events . . . 520
36.6 Graph Transformations . . . 520
36.6.1 The Layout Pipeline . . . 520
36.6.2 Hints For Edge Routing . . . 523
36.7 The Interface To Algorithms . . . 523
36.8 Examples of Implementations of Graph Drawing Algorithms . . . 530
36.8.1 The “Hello World” of Graph Drawing . . . 530
36.8.2 How To Generate Edges Inside an Algorithm . . . 531
36.8.3 How To Generate Nodes Inside an Algorithm . . . 532
36.9 Support Libraries . . . 534
36.9.1 Basic Functions . . . 534
36.9.2 Lookup Tables . . . 537
36.9.3 Computing Distances in Graphs . . . 538
36.9.4 Priority Queues . . . 538

37 Writing Graph Drawing Algorithms in C **539**
37.1 How C and TEX Communicate . . . 539
37.2 Writing Graph Drawing Algorithms in C . . . 540
37.2.1 The Hello World of Graph Drawing in C . . . 540
37.2.2 Documenting Algorithms Written in C . . . 542
37.2.3 The Interface From C . . . 542
37.3 Writing Graph Drawing Algorithms in C++ . . . 543
37.3.1 The Hello World of Graph Drawing in C++ . . . 543
37.3.2 The Interface From C++ . . . 544
37.4 Writing Graph Drawing Algorithms Using OGDF . . . 546
37.4.1 The Hello World of Graph Drawing in OGDF – From Scratch . . . 546
37.4.2 The Hello World of Graph Drawing in OGDF – Adapting Existing Classes . . . 547
37.4.3 Documenting OGDF Algorithms . . . 548
37.4.4 The Interface From OGDF . . . 548

38 The Display Layer **549**
38.1 Introduction: The Interplay of the Different Layers . . . 549
38.2 An Example Display System . . . 550
38.3 The Interface to Display Systems . . . 551

39 The Binding Layer **557**
39.1 Overview . . . 557
39.2 The Binding Class and the Interface Core . . . 557
39.3 The Binding To PGF . . . 560
39.4 An Example Binding Class . . . 560

V Libraries **563**

40 Three Dimensional Drawing Library **564**
40.1 Coordinate Systems . . . 564
40.2 Coordinate Planes . . . 564
40.2.1 Switching to an arbitrary plane . . . 565
40.2.2 Predefined planes . . . 565
40.3 Examples . . . 566

41 Angle Library **568**

42 Arrow Tip Library **570**

43 Automata Drawing Library **571**
43.1 Drawing Automata . . . 571
43.2 States With and Without Output . . . 572
43.3 Initial and Accepting States . . . 572
43.4 Examples . . . 574

44 Babel Library **576**

45 Background Library **577**

46 Calc Library **581**

47 Calendar Library **582**
47.1 Calendar Command . . . 582
47.1.1 Creating a Simple List of Days . . . 589
47.1.2 Adding a Month Label . . . 589
47.1.3 Creating a Week List Arrangement . . . 589
47.1.4 Creating a Month List Arrangement . . . 590
47.2 Arrangements . . . 590
47.3 Month Labels . . . 593

47.4 Examples . . . 595

48 Chains **600**
48.1 Overview . . . 600
48.2 Starting and Continuing a Chain . . . 600
48.3 Nodes on a Chain . . . 602
48.4 Joining Nodes on a Chain . . . 604
48.5 Branches . . . 605

49 Circuit Libraries **607**
49.1 Introduction . . . 607
49.1.1 A First Example . . . 607
49.1.2 Symbols . . . 608
49.1.3 Symbol Graphics . . . 608
49.1.4 Annotations . . . 609
49.2 The Base Circuit Library . . . 609
49.2.1 Symbol Size . . . 610
49.2.2 Declaring New Symbols . . . 610
49.2.3 Pointing Symbols in the Right Direction . . . 612
49.2.4 Info Labels . . . 613
49.2.5 Declaring and Using Annotations . . . 614
49.2.6 Theming Symbols . . . 615
49.3 Logical Circuits . . . 617
49.3.1 Overview . . . 617
49.3.2 Symbols: The Gates . . . 620
49.3.3 Implementation: The Logic Gates Shape Library . . . 620
49.3.4 Implementation: The US-Style Logic Gates Shape Library . . . 622
49.3.5 Implementation: The IEC-Style Logic Gates Shape Library . . . 623
49.4 Electrical Engineering Circuits . . . 625
49.4.1 Overview . . . 625
49.4.2 Symbols: Indicating Current Directions . . . 629
49.4.3 Symbols: Basic Elements . . . 629
49.4.4 Symbols: Diodes . . . 629
49.4.5 Symbols: Contacts . . . 629
49.4.6 Symbols: Measurement devices . . . 629
49.4.7 Units . . . 630
49.4.8 Annotations . . . 630
49.4.9 Implementation: The EE-Symbols Shape Library . . . 630
49.4.10 Implementation: The IEC-Style EE-Symbols Shape Library . . . 632

50 Decoration Library **637**
50.1 Overview and Common Options . . . 637
50.2 Handling “Dimension too large” errors . . . 638
50.3 Path Morphing Decorations . . . 638
50.3.1 Decorations Producing Straight Line Paths . . . 638
50.3.2 Decorations Producing Curved Line Paths . . . 640
50.4 Path Replacing Decorations . . . 641
50.5 Marking Decorations . . . 644
50.5.1 Overview . . . 644
50.6 Arbitrary Markings . . . 644
50.6.1 Arrow Tip Markings . . . 649
50.6.2 Footprint Markings . . . 649
50.6.3 Shape Background Markings . . . 650
50.7 Text Decorations . . . 654
50.8 Fractal Decorations . . . 664

51 Entity-Relationship Diagram Drawing Library **667**
51.1 Entities . . . 667
51.2 Relationships . . . 667
51.3 Attributes . . . 668

52 Externalization Library **669**
52.1 Overview . . . 669
52.2 Requirements . . . 669
52.3 A Word About ConTEXt And Plain TEX . . . 669
52.4 Externalizing Graphics . . . 669
52.4.1 Support for Labels and References In External Files . . . 671
52.4.2 Customizing the Generated File Names . . . 672
52.4.3 Remaking Figures or Skipping Figures . . . 673
52.4.4 Customizing the Externalization . . . 676
52.4.5 Details About The Process . . . 679
52.5 Using External Graphics Without PGF Installed . . . 680
52.6 eps Graphics Export . . . 680
52.7 Bitmap Graphics Export . . . 681
52.8 Compatibility Issues . . . 681
52.8.1 References In External Pictures . . . 681
52.8.2 Compatibility With Other Libraries or Packages . . . 682
52.8.3 Compatibility With Bounding Box Restrictions . . . 682
52.8.4 Interoperability With The Basic Layer Externalization . . . 682

53 Fading Library **683**

54 Fitting Library **684**

55 Fixed Point Arithmetic Library **687**
55.1 Overview . . . 687
55.2 Using Fixed Point Arithmetic in PGF and Ti*k*Z . . . 687

56 Floating Point Unit Library **689**
56.1 Overview . . . 689
56.2 Usage . . . 689
56.3 Comparison to the fixed point arithmetic library . . . 690
56.4 Command Reference and Programmer's Manual . . . 691
56.4.1 Creating and Converting Floats . . . 691
56.4.2 Symbolic Rounding Operations . . . 694
56.4.3 Math Operations Commands . . . 695
56.4.4 Accessing the Original Math Routines for Programmers . . . 697

57 Lindenmayer System Drawing Library **698**
57.1 Overview . . . 698
57.1.1 Declaring L-systems . . . 698
57.2 Using Lindenmayer Systems . . . 700
57.2.1 Using L-Systems in PGF . . . 700
57.2.2 Using L-Systems in Ti*k*Z . . . 701

58 Math Library **703**
58.1 Overview . . . 703
58.2 Assignment . . . 704
58.3 Integers, "Real" Numbers, and Coordinates . . . 705
58.4 Repeating Things . . . 706
58.5 Branching Statements . . . 707
58.6 Declaring Functions . . . 707
58.7 Executing Code Outside the Parser . . . 708

59 Matrix Library **709**
59.1 Matrices of Nodes . . . 709
59.2 End-of-Lines and End-of-Row Characters in Matrices of Nodes . . . 710
59.3 Delimiters . . . 711

60 Mindmap Drawing Library **713**
60.1 Overview . . . 713
60.2 The Mindmap Style . . . 713
60.3 Concepts Nodes . . . 714
60.3.1 Isolated Concepts . . . 714
60.3.2 Concepts in Trees . . . 715
60.4 Connecting Concepts . . . 717
60.4.1 Simple Connections . . . 717
60.4.2 The Circle Connection Bar Decoration . . . 718
60.4.3 The Circle Connection Bar To-Path . . . 719
60.4.4 Tree Edges . . . 720
60.5 Adding Annotations . . . 721

61 Paper-Folding Diagrams Library **723**

62 Pattern Library **730**
62.1 Form-Only Patterns . . . 730
62.2 Inherently Colored Patterns . . . 731
62.3 User-Defined Patterns . . . 731

63 Three Point Perspective Drawing Library **738**
63.1 Coordinate Systems . . . 738
63.2 Setting the view . . . 738
63.3 Defining the perspective . . . 739
63.4 Shortcomings . . . 741
63.5 Examples . . . 741

64 Petri-Net Drawing Library **744**
64.1 Places . . . 744
64.2 Transitions . . . 744
64.3 Tokens . . . 745
64.4 Examples . . . 747

65 Plot Handler Library **749**
65.1 Curve Plot Handlers . . . 749
65.2 Constant Plot Handlers . . . 750
65.3 Comb Plot Handlers . . . 752
65.4 Bar Plot Handlers . . . 753
65.5 Gapped Plot Handlers . . . 755
65.6 Mark Plot Handler . . . 755

66 Plot Mark Library **758**

67 Profiler Library **760**
67.1 Overview . . . 760
67.2 Requirements . . . 760
67.3 Defining Profiler Entries . . . 760

68 Resource Description Framework Library **763**
68.1 Starting the RDF Engine . . . 763
68.2 Creating Statements . . . 764
68.3 Creating Resources . . . 766
68.4 Creating Containers . . . 767
68.5 Creating Semantic Information Inside Styles and Libraries . . . 769
68.5.1 An Example Library for Drawing Finite Automata . . . 769

68.5.2 Adding Semantic Information About the Automata as a Whole . . . 769
68.5.3 Adding Semantic Information About the States . . . 770
68.5.4 Adding Semantic Information About the Transitions . . . 771
68.5.5 Using Containers . . . 771
68.5.6 The Resulting RDF Graph . . . 772

69 Shadings Library 776

70 Shadows Library 780
70.1 Overview . . . 780
70.2 The General Shadow Option . . . 780
70.3 Shadows for Arbitrary Paths and Shapes . . . 781
70.3.1 Drop Shadows . . . 781
70.3.2 Copy Shadows . . . 781
70.4 Shadows for Special Paths and Nodes . . . 782

71 Shape Library 785
71.1 Overview . . . 785
71.2 Predefined Shapes . . . 785
71.3 Geometric Shapes . . . 786
71.4 Symbol Shapes . . . 801
71.5 Arrow Shapes . . . 810
71.6 Shapes with Multiple Text Parts . . . 816
71.7 Callout Shapes . . . 823
71.8 Miscellaneous Shapes . . . 827

72 Spy Library: Magnifying Parts of Pictures 832
72.1 Magnifying a Part of a Picture . . . 832
72.2 Spy Scopes . . . 833
72.3 The Spy Command . . . 833
72.4 Predefined Spy Styles . . . 835
72.5 Examples . . . 836

73 SVG-Path Library 838

74 To Path Library 839
74.1 Straight Lines . . . 839
74.2 Move-Tos . . . 839
74.3 Curves . . . 839
74.4 Loops . . . 842

75 Through Library 844

76 Tree Library 845
76.1 Growth Functions . . . 845
76.2 Edges From Parent . . . 847

77 Turtle Graphics Library 848

78 Views Library 850

VI Data Visualization 852

79 Introduction to Data Visualization 853
79.1 Concept: Data Points . . . 853
79.2 Concept: Visualization Pipeline . . . 853

80 Creating Data Visualizations **855**
80.1 Overview . . . 855
80.2 Concept: Data Points and Data Formats . . . 855
80.3 Concept: Axes, Ticks, and Grids . . . 856
80.4 Concept: Visualizers . . . 857
80.5 Concept: Style Sheets and Legends . . . 857
80.6 Usage . . . 858
80.7 Advanced: Executing User Code During a Data Visualization . . . 863
80.8 Advanced: Creating New Objects . . . 863

81 Providing Data for a Data Visualization **865**
81.1 Overview . . . 865
81.2 Concepts . . . 865
81.3 Reference: Built-In Formats . . . 865
81.4 Reference: Advanced Formats . . . 867
81.5 Advanced: The Data Parsing Process . . . 869
81.6 Advanced: Defining New Formats . . . 871

82 Axes **873**
82.1 Overview . . . 873
82.2 Basic Configuration of Axes . . . 873
82.2.1 Usage . . . 873
82.2.2 The Axis Attribute . . . 874
82.2.3 The Axis Attribute Range Interval . . . 875
82.2.4 Scaling: The General Mechanism . . . 876
82.2.5 Scaling: Logarithmic Axes . . . 878
82.2.6 Scaling: Setting the Length or Unit Length . . . 879
82.2.7 Axis Label . . . 880
82.2.8 Reference: Axis Types . . . 881
82.3 Axis Systems . . . 881
82.3.1 Usage . . . 881
82.3.2 Reference: Scientific Axis Systems . . . 882
82.3.3 Reference: School Book Axis Systems . . . 885
82.3.4 Advanced Reference: Underlying Cartesian Axis Systems . . . 887
82.4 Ticks and Grids . . . 888
82.4.1 Concepts . . . 888
82.4.2 The Main Options: Tick and Grid . . . 888
82.4.3 Semi-Automatic Computation of Tick and Grid Line Positions . . . 889
82.4.4 Automatic Computation of Tick and Grid Line Positions . . . 891
82.4.5 Manual Specification of Tick and Grid Line Positions . . . 894
82.4.6 Styling Ticks and Grid Lines: Introduction . . . 896
82.4.7 Styling Ticks and Grid Lines: The Style and Node Style Keys . . . 896
82.4.8 Styling Ticks and Grid Lines: Styling Grid Lines . . . 897
82.4.9 Styling Ticks and Grid Lines: Styling Ticks and Tick Labels . . . 898
82.4.10 Styling Ticks and Grid Lines: Exceptional Ticks . . . 900
82.4.11 Styling Ticks and Grid Lines: Styling and Typesetting a Value . . . 900
82.4.12 Stacked Ticks . . . 902
82.4.13 Reference: Basic Strategies . . . 905
82.4.14 Advanced: Defining New Placement Strategies . . . 906
82.5 Advanced: Creating New Axis Systems . . . 906
82.5.1 Creating the Axes . . . 907
82.5.2 Visualizing the Axes . . . 908
82.5.3 Visualizing Grid Lines . . . 911
82.5.4 Visualizing the Ticks and Tick Labels . . . 913
82.5.5 Visualizing the Axis Labels . . . 916
82.5.6 The Complete Axis System . . . 917
82.5.7 Using the New Axis System Key . . . 919

83 Visualizers **921**
83.1 Overview . . . 921
83.2 Usage . . . 921
83.2.1 Using a Single Visualizer . . . 921
83.2.2 Using Multiple Visualizers . . . 922
83.2.3 Styling a Visualizer . . . 923
83.3 Reference: Basic Visualizers . . . 925
83.3.1 Visualizing Data Points Using Lines . . . 925
83.3.2 Visualizing Data Points Using Marks . . . 928
83.4 Advanced: Creating New Visualizers . . . 928

84 Style Sheets and Legends **931**
84.1 Overview . . . 931
84.2 Concepts: Style Sheets . . . 931
84.3 Concepts: Legends . . . 932
84.4 Usage: Style Sheets . . . 933
84.4.1 Picking a Style Sheet . . . 933
84.4.2 Creating a New Style Sheet . . . 934
84.4.3 Creating a New Color Style Sheet . . . 936
84.5 Reference: Style Sheets for Lines . . . 936
84.6 Reference: Style Sheets for Scatter Plots . . . 938
84.7 Reference: Color Style Sheets . . . 939
84.8 Usage: Labeling Data Sets Inside the Visualization . . . 941
84.8.1 Placing a Label Next to a Data Set . . . 941
84.8.2 Connecting a Label to a Data Set via a Pin . . . 943
84.9 Usage: Labeling Data Sets Inside a Legend . . . 944
84.9.1 Creating Legends and Legend Entries . . . 945
84.9.2 Rows and Columns of Legend Entries . . . 947
84.9.3 Legend Placement: The General Mechanism . . . 951
84.9.4 Legend Placement: Outside to the Data Visualization . . . 951
84.9.5 Legend Placement: Inside to the Data Visualization . . . 953
84.9.6 Legend Entries: General Styling . . . 955
84.9.7 Legend Entries: Styling the Text Node . . . 956
84.9.8 Legend Entries: Text Placement . . . 957
84.9.9 Advanced: Labels in Legends and Their Visualizers . . . 957
84.9.10 Reference: Label in Legend Visualizers for Lines and Scatter Plots . . . 959

85 Polar Axes **964**
85.1 Overview . . . 964
85.2 Scientific Polar Axis System . . . 965
85.2.1 Tick Placements . . . 966
85.2.2 Angle Ranges . . . 967
85.3 Advanced: Creating a New Polar Axis System . . . 971

86 The Data Visualization Backend **973**
86.1 Overview . . . 973
86.2 The Rendering Pipeline . . . 973
86.3 Usage . . . 973
86.4 The Mathematical Micro-Kernel . . . 973

VII Utilities 974

87 Key Management **975**
87.1 Introduction . . . 975
87.1.1 Comparison to Other Packages . . . 975
87.1.2 Quick Guide to Using the Key Mechanism . . . 975
87.2 The Key Tree . . . 976
87.3 Setting Keys . . . 978

87.3.1 First Char Syntax Detection . . . 979
87.3.2 Default Arguments . . . 981
87.3.3 Keys That Execute Commands . . . 981
87.3.4 Keys That Store Values . . . 982
87.3.5 Keys That Are Handled . . . 983
87.3.6 Keys That Are Unknown . . . 984
87.3.7 Search Paths And Handled Keys . . . 985
87.4 Key Handlers . . . 985
87.4.1 Handlers for Path Management . . . 986
87.4.2 Setting Defaults . . . 986
87.4.3 Defining Key Codes . . . 986
87.4.4 Defining Styles . . . 988
87.4.5 Defining Value-, Macro-, If- and Choice-Keys . . . 989
87.4.6 Expanded and Multiple Values . . . 991
87.4.7 Handlers for Forwarding . . . 992
87.4.8 Handlers for Testing Keys . . . 994
87.4.9 Handlers for Key Inspection . . . 994
87.5 Error Keys . . . 995
87.6 Key Filtering . . . 995
87.6.1 Starting With An Example . . . 996
87.6.2 Setting Filters . . . 996
87.6.3 Handlers For Unprocessed Keys . . . 998
87.6.4 Family Support . . . 998
87.6.5 Other Key Filters . . . 1000
87.6.6 Programmer Interface . . . 1000
87.6.7 Defining Own Filters Or Filter Handlers . . . 1001

88 Repeating Things: The Foreach Statement 1003

89 Date and Calendar Utility Macros 1009
89.1 Handling Dates . . . 1009
89.1.1 Conversions Between Date Types . . . 1009
89.1.2 Checking Dates . . . 1010
89.1.3 Typesetting Dates . . . 1011
89.1.4 Localization . . . 1012
89.2 Typesetting Calendars . . . 1012

90 Page Management 1016
90.1 Basic Usage . . . 1016
90.2 The Predefined Layouts . . . 1017
90.3 Defining a Layout . . . 1019
90.4 Creating Logical Pages . . . 1022

91 Extended Color Support 1023

92 Parser Module 1024
92.1 Keys of the Parser Module . . . 1026
92.2 Examples . . . 1026

VIII Mathematical and Object-Oriented Engines 1028

93 Design Principles 1029
93.1 Loading the Mathematical Engine . . . 1029
93.2 Layers of the Mathematical Engine . . . 1029
93.3 Efficiency and Accuracy of the Mathematical Engine . . . 1029

94 Mathematical Expressions **1030**
94.1 Parsing Expressions . . . 1030
94.1.1 Commands . . . 1030
94.1.2 Considerations Concerning Units . . . 1032
94.2 Syntax for Mathematical Expressions: Operators . . . 1033
94.3 Syntax for Mathematical Expressions: Functions . . . 1035
94.3.1 Basic arithmetic functions . . . 1035
94.3.2 Rounding functions . . . 1038
94.3.3 Integer arithmetic functions . . . 1039
94.3.4 Trigonometric functions . . . 1039
94.3.5 Comparison and logical functions . . . 1042
94.3.6 Pseudo-random functions . . . 1043
94.3.7 Base conversion functions . . . 1044
94.3.8 Miscellaneous functions . . . 1045

95 Additional Mathematical Commands **1047**
95.1 Basic arithmetic functions . . . 1047
95.2 Comparison and logical functions . . . 1047
95.3 Pseudo-Random Numbers . . . 1047
95.4 Base Conversion . . . 1048
95.5 Angle Computations . . . 1049

96 Customizing the Mathematical Engine **1050**

97 Number Printing **1053**
97.1 Changing display styles . . . 1058

98 Object-Oriented Programming **1064**
98.1 Overview . . . 1064
98.2 A Running Example: The Stamp Class . . . 1064
98.3 Classes . . . 1064
98.4 Objects . . . 1065
98.5 Methods . . . 1066
98.6 Attributes . . . 1067
98.7 Identities . . . 1069
98.8 The Object Class . . . 1070
98.9 The Signal Class . . . 1070
98.10 Implementation Notes . . . 1071

IX The Basic Layer 1072

99 Design Principles **1073**
99.1 Core and Modules . . . 1073
99.2 Communicating with the Basic Layer via Macros . . . 1073
99.3 Path-Centered Approach . . . 1074
99.4 Coordinate Versus Canvas Transformations . . . 1074

100 Hierarchical Structures: Package, Environments, Scopes, and Text **1075**
100.1 Overview . . . 1075
100.1.1 The Hierarchical Structure of the Package . . . 1075
100.1.2 The Hierarchical Structure of Graphics . . . 1075
100.2 The Hierarchical Structure of the Package . . . 1076
100.2.1 The Core Package . . . 1076
100.2.2 The Modules . . . 1077
100.2.3 The Library Packages . . . 1077
100.3 The Hierarchical Structure of the Graphics . . . 1077
100.3.1 The Main Environment . . . 1077
100.3.2 Graphic Scope Environments . . . 1079

100.3.3 Inserting Text and Images . . . 1082
100.4 Object Identifiers . . . 1084
100.4.1 Commands for Creating Graphic Objects . . . 1084
100.4.2 Settings and Querying Identifiers . . . 1085
100.5 Resource Description Framework Annotations (RDFa) . . . 1086
100.6 Error Messages and Warnings . . . 1087

101 Specifying Coordinates 1088
101.1 Overview . . . 1088
101.2 Basic Coordinate Commands . . . 1088
101.3 Coordinates in the XY-Coordinate System . . . 1088
101.4 Three Dimensional Coordinates . . . 1089
101.5 Building Coordinates From Other Coordinates . . . 1090
101.5.1 Basic Manipulations of Coordinates . . . 1090
101.5.2 Points Traveling along Lines and Curves . . . 1091
101.5.3 Points on Borders of Objects . . . 1093
101.5.4 Points on the Intersection of Lines . . . 1093
101.5.5 Points on the Intersection of Two Circles . . . 1094
101.5.6 Points on the Intersection of Two Paths . . . 1094
101.6 Extracting Coordinates . . . 1095
101.7 Internals of How Point Commands Work . . . 1095

102 Constructing Paths 1097
102.1 Overview . . . 1097
102.2 The Move-To Path Operation . . . 1097
102.3 The Line-To Path Operation . . . 1098
102.4 The Curve-To Path Operations . . . 1098
102.5 The Close Path Operation . . . 1099
102.6 Arc, Ellipse and Circle Path Operations . . . 1100
102.7 Rectangle Path Operations . . . 1103
102.8 The Grid Path Operation . . . 1103
102.9 The Parabola Path Operation . . . 1104
102.10 Sine and Cosine Path Operations . . . 1104
102.11 Plot Path Operations . . . 1105
102.12 Rounded Corners . . . 1105
102.13 Internal Tracking of Bounding Boxes for Paths and Pictures . . . 1106

103 Decorations 1108
103.1 Overview . . . 1108
103.2 Decoration Automata . . . 1108
103.2.1 The Different Paths . . . 1108
103.2.2 Segments and States . . . 1109
103.3 Declaring Decorations . . . 1110
103.3.1 Predefined Decorations . . . 1114
103.4 Using Decorations . . . 1114
103.5 Meta-Decorations . . . 1117
103.5.1 Declaring Meta-Decorations . . . 1118
103.5.2 Predefined Meta-decorations . . . 1119
103.5.3 Using Meta-Decorations . . . 1119

104 Using Paths 1121
104.1 Overview . . . 1121
104.2 Stroking a Path . . . 1122
104.2.1 Graphic Parameter: Line Width . . . 1122
104.2.2 Graphic Parameter: Caps and Joins . . . 1122
104.2.3 Graphic Parameter: Dashing . . . 1122
104.2.4 Graphic Parameter: Stroke Color . . . 1123
104.2.5 Graphic Parameter: Stroke Opacity . . . 1123
104.2.6 Inner Lines . . . 1123

104.3 Arrow Tips on a Path . . . 1124
104.4 Filling a Path . . . 1125
104.4.1 Graphic Parameter: Interior Rule . . . 1125
104.4.2 Graphic Parameter: Filling Color . . . 1126
104.4.3 Graphic Parameter: Fill Opacity . . . 1126
104.5 Clipping a Path . . . 1126
104.6 Using a Path as a Bounding Box . . . 1126

105 Defining New Arrow Tip Kinds 1127
105.1 Overview . . . 1127
105.2 Terminology . . . 1127
105.3 Caching and Rendering of Arrows . . . 1128
105.4 Declaring an Arrow Tip Kind . . . 1129
105.5 Handling Arrow Options . . . 1133
105.5.1 Dimension Options . . . 1133
105.5.2 True–False Options . . . 1134
105.5.3 Inaccessible Options . . . 1134
105.5.4 Defining New Arrow Keys . . . 1135

106 Nodes and Shapes 1137
106.1 Overview . . . 1137
106.1.1 Creating and Referencing Nodes . . . 1137
106.1.2 Anchors . . . 1137
106.1.3 Layers of a Shape . . . 1137
106.1.4 Node Parts . . . 1138
106.2 Creating Nodes . . . 1138
106.2.1 Creating Simple Nodes . . . 1138
106.2.2 Creating Multi-Part Nodes . . . 1139
106.2.3 Deferred Node Positioning . . . 1141
106.3 Using Anchors . . . 1142
106.3.1 Referencing Anchors of Nodes in the Same Picture . . . 1143
106.3.2 Referencing Anchors of Nodes in Different Pictures . . . 1144
106.4 Special Nodes . . . 1144
106.5 Declaring New Shapes . . . 1145
106.5.1 What Must Be Defined For a Shape? . . . 1145
106.5.2 Normal Anchors Versus Saved Anchors . . . 1146
106.5.3 Command for Declaring New Shapes . . . 1146

107 Matrices 1152
107.1 Overview . . . 1152
107.2 Cell Pictures and Their Alignment . . . 1152
107.3 The Matrix Command . . . 1152
107.4 Row and Column Spacing . . . 1154
107.5 Callbacks . . . 1155

108 Coordinate, Canvas, and Nonlinear Transformations 1157
108.1 Overview . . . 1157
108.2 Coordinate Transformations . . . 1157
108.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix . . . 1157
108.2.2 Commands for Relative Coordinate Transformations . . . 1158
108.2.3 Commands for Absolute Coordinate Transformations . . . 1161
108.2.4 Saving and Restoring the Coordinate Transformation Matrix . . . 1162
108.2.5 Applying Coordinate Transformation to Points . . . 1163
108.2.6 Computing Adjustments for Coordinate Transformations . . . 1163
108.3 Canvas Transformations . . . 1164
108.3.1 Applying General Canvas Transformations . . . 1164
108.3.2 Establishing View Boxes . . . 1165
108.4 Nonlinear Transformations . . . 1166
108.4.1 Introduction . . . 1166

108.4.2 Installing Nonlinear Transformation . . . 1166
108.4.3 Applying Nonlinear Transformations to Points . . . 1167
108.4.4 Applying Nonlinear Transformations to Paths . . . 1167
108.4.5 Applying Nonlinear Transformations to Text . . . 1168
108.4.6 Approximating Nonlinear Transformations Using Linear Transformations . . . 1169
108.4.7 Nonlinear Transformation Libraries . . . 1169

109 Patterns 1172
109.1 Overview . . . 1172
109.2 Declaring a Pattern . . . 1172
109.3 Setting a Pattern . . . 1174

110 Declaring and Using Images 1175
110.1 Overview . . . 1175
110.2 Declaring an Image . . . 1175
110.3 Using an Image . . . 1176
110.4 Masking an Image . . . 1177

111 Externalizing Graphics 1179
111.1 Overview . . . 1179
111.2 Workflow Step 1: Naming Graphics . . . 1179
111.3 Workflow Step 2: Generating the External Graphics . . . 1180
111.4 Workflow Step 3: Including the External Graphics . . . 1181
111.5 A Complete Example . . . 1182

112 Creating Plots 1186
112.1 Overview . . . 1186
112.2 Generating Plot Streams . . . 1186
112.2.1 Basic Building Blocks of Plot Streams . . . 1186
112.2.2 Commands That Generate Plot Streams . . . 1188
112.3 Plot Handlers . . . 1190
112.4 Defining New Plot Handlers . . . 1191

113 Layered Graphics 1193
113.1 Overview . . . 1193
113.2 Declaring Layers . . . 1193
113.3 Using Layers . . . 1193

114 Shadings 1195
114.1 Overview . . . 1195
114.1.1 Color models . . . 1195
114.2 Declaring Shadings . . . 1196
114.2.1 Horizontal and Vertical Shadings . . . 1196
114.2.2 Radial Shadings . . . 1196
114.2.3 General (Functional) Shadings . . . 1197
114.3 Using Shadings . . . 1201

115 Transparency 1204
115.1 Specifying a Uniform Opacity . . . 1204
115.2 Specifying a Blend Mode . . . 1204
115.3 Specifying a Fading . . . 1205
115.4 Transparency Groups . . . 1207

116 Animations 1209
116.1 Overview . . . 1209
116.2 Animating an Attribute . . . 1209
116.2.1 The Main Command . . . 1209
116.2.2 Specifying the Timeline . . . 1211
116.2.3 “Anti-Animations”: Snapshots . . . 1214
116.3 Animating Color, Opacity, Visibility, and Staging . . . 1215

116.4 Animating Paths and their Rendering . . . 1218
116.5 Animating Transformations and Views . . . 1221
116.6 Commands for Specifying Timing: Beginnings and Endings . . . 1224
116.7 Commands for Specifying Timing: Repeats . . . 1226

117 Adding libraries to pgf: temporary registers 1228

118 Quick Commands 1230
118.1 Quick Coordinate Commands . . . 1230
118.2 Quick Path Construction Commands . . . 1230
118.3 Quick Path Usage Commands . . . 1231
118.4 Quick Text Box Commands . . . 1231

X The System Layer 1233

119 Design of the System Layer 1234
119.1 Driver Files . . . 1234
119.2 Common Definition Files . . . 1234

120 Commands of the System Layer 1235
120.1 Beginning and Ending a Stream of System Commands . . . 1235
120.2 Scoping System Commands . . . 1236
120.3 Path Construction System Commands . . . 1236
120.4 Canvas Transformation System Commands . . . 1237
120.5 Stroking, Filling, and Clipping System Commands . . . 1238
120.6 Graphic State Option System Commands . . . 1239
120.7 Color System Commands . . . 1240
120.8 Pattern System Commands . . . 1242
120.9 Image System Commands . . . 1242
120.10 Shading System Commands . . . 1243
120.11 Transparency System Commands . . . 1244
120.12 Animation Commands . . . 1245
120.13 Object Identification System Commands . . . 1245
120.14 Resource Description Framework Annotations (RDFa) . . . 1246
120.15 Reusable Objects System Commands . . . 1247
120.16 Invisibility System Commands . . . 1248
120.17 Page Size Commands . . . 1248
120.18 Position Tracking Commands . . . 1249
120.19 Internal Conversion Commands . . . 1249

121 The Soft Path Subsystem 1251
121.1 Path Creation Process . . . 1251
121.2 Starting and Ending a Soft Path . . . 1251
121.3 Soft Path Creation Commands . . . 1252
121.4 The Soft Path Data Structure . . . 1252

122 The Protocol Subsystem 1254

123 Animation System Layer 1255
123.1 Animations and Snapshots . . . 1255
123.2 Commands for Animating an Attribute: Color, Opacity, Visibility, Staging . . . 1257
123.3 Commands for Animating an Attribute: Paths and Their Rendering . . . 1258
123.4 Commands for Animating an Attribute: Transformations and Views . . . 1261
123.5 Commands for Specifying the Target Object . . . 1265
123.6 Commands for Specifying Timelines: Specifying Times . . . 1265
123.7 Commands for Specifying Timelines: Specifying Values . . . 1266
123.8 Commands for Specifying Timing: Repeats . . . 1268
123.9 Commands for Specifying Timing: Beginning and Ending . . . 1268
123.10 Commands for Specifying Timing: Restart Behaviour . . . 1270

123.11 Commands for Specifying Accumulation 1272

XI References and Index **1273**

Index **1274**

1 Introduction

Welcome to the documentation of Ti*k*Z and the underlying PGF system. What began as a small LaTeX style for creating the graphics in my (Till Tantau's) PhD thesis directly with pdfLaTeX has now grown to become a full-blown graphics language with a manual of over a thousand pages. The wealth of options offered by Ti*k*Z is often daunting to beginners; but fortunately this documentation comes with a number of slowly-paced tutorials that will teach you almost all you should know about Ti*k*Z without your having to read the rest.

I wish to start with the question "What is Ti*k*Z?" Basically, it just defines a number of TeX commands that draw graphics. For example, the code `\tikz \draw (0pt,0pt) -- (20pt,6pt);` yields the line  and the code `\tikz \fill[orange] (1ex,1ex) circle [radius=1ex];` yields . In a sense, when you use Ti*k*Z you "program" your graphics, just as you "program" your document when you use TeX. This also explains the name: Ti*k*Z is a recursive acronym in the tradition of "GNU's Not Unix" and means "Ti*k*Z ist *kein* Zeichenprogramm", which translates to "Ti*k*Z is not a drawing program", cautioning the reader as to what to expect. With Ti*k*Z you get all the advantages of the "TeX-approach to typesetting" for your graphics: quick creation of simple graphics, precise positioning, the use of macros, often superior typography. You also inherit all the disadvantages: steep learning curve, no WYSIWYG, small changes require a long recompilation time, and the code does not really "show" how things will look like.

Now that we know what Ti*k*Z is, what about "PGF"? As mentioned earlier, Ti*k*Z started out as a project to implement TeX graphics macros that can be used both with pdfLaTeX and also with the classical (PostScript-based) LaTeX. In other words, I wanted to implement a "portable graphics format" for TeX – hence the name PGF. These early macros are still around and they form the "basic layer" of the system described in this manual, but most of the interaction an author has these days is with Ti*k*Z – which has become a whole language of its own.

1.1 The Layers Below Ti*k*Z

It turns out that there are actually *two* layers below Ti*k*Z:

System layer: This layer provides a complete abstraction of what is going on "in the driver". The driver is a program like `dvips` or `dvipdfm` that takes a `.dvi` file as input and generates a `.ps` or a `.pdf` file. (The `pdftex` program also counts as a driver, even though it does not take a `.dvi` file as input. Never mind.) Each driver has its own syntax for the generation of graphics, causing headaches to everyone who wants to create graphics in a portable way. PGF's system layer "abstracts away" these differences. For example, the system command `\pgfsys@lineto{10pt}{10pt}` extends the current path to the coordinate $(10\text{pt}, 10\text{pt})$ of the current `{pgfpicture}`. Depending on whether `dvips`, `dvipdfm`, or `pdftex` is used to process the document, the system command will be converted to different `\special` commands. The system layer is as "minimalistic" as possible since each additional command makes it more work to port PGF to a new driver.

As a user, you will not use the system layer directly.

Basic layer: The basic layer provides a set of basic commands that allow you to produce complex graphics in a much easier manner than by using the system layer directly. For example, the system layer provides no commands for creating circles since circles can be composed from the more basic Bézier curves (well, almost). However, as a user you will want to have a simple command to create circles (at least I do) instead of having to write down half a page of Bézier curve support coordinates. Thus, the basic layer provides a command `\pgfpathcircle` that generates the necessary curve coordinates for you.

The basic layer consists of a *core*, which consists of several interdependent packages that can only be loaded *en bloc*, and additional *modules* that extend the core by more special-purpose commands like node management or a plotting interface. For instance, the BEAMER package uses only the core and not, say, the `shapes` modules.

In theory, Ti*k*Z itself is just one of several possible "frontends". which are sets of commands or a special syntax that makes using the basic layer easier. A problem with directly using the basic layer is that code written for this layer is often too "verbose". For example, to draw a simple triangle, you may need as many as five commands when using the basic layer: One for beginning a path at the first corner of the triangle, one for extending the path to the second corner, one for going to the third, one for closing the path, and one for actually painting the triangle (as opposed to filling it). With the Ti*k*Z frontend all this boils down to a single simple METAFONT-like command:

```
\draw (0,0) -- (1,0) -- (1,1) -- cycle;
```

In practice, Ti*k*Z is the only "serious" frontend for PGF. It gives you access to all features of PGF, but it is intended to be easy to use. The syntax is a mixture of METAFONT and PSTRICKS and some ideas of myself. There are other frontends besides Ti*k*Z, but they are intended more as "technology studies" and less as serious alternatives to Ti*k*Z. In particular, the `pgfpict2e` frontend reimplements the standard LaTeX `{picture}` environment and commands like `\line` or `\vector` using the PGF basic layer. This layer is not really "necessary" since the `pict2e.sty` package does at least as good a job at reimplementing the `{picture}` environment. Rather, the idea behind this package is to have a simple demonstration of how a frontend can be implemented.

Since most users will only use Ti*k*Z and almost no one will use the system layer directly, this manual is mainly about Ti*k*Z in the first parts; the basic layer and the system layer are explained at the end.

1.2 Comparison with Other Graphics Packages

Ti*k*Z is not the only graphics package for TeX. In the following, I try to give a reasonably fair comparison of Ti*k*Z and other packages.

1. The standard LaTeX `{picture}` environment allows you to create simple graphics, but little more. This is certainly not due to a lack of knowledge or imagination on the part of LaTeX's designer(s). Rather, this is the price paid for the `{picture}` environment's portability: It works together with all backend drivers.

2. The `pstricks` package is certainly powerful enough to create any conceivable kind of graphic, but it is not really portable. Most importantly, it does not work with `pdftex` nor with any other driver that produces anything but PostScript code.

   Compared to Ti*k*Z, `pstricks` has a similar support base. There are many nice extra packages for special purpose situations that have been contributed by users over the last decade. The Ti*k*Z syntax is more consistent than the `pstricks` syntax as Ti*k*Z was developed "in a more centralized manner" and also "with the shortcomings on `pstricks` in mind".

3. The `xypic` package is an older package for creating graphics. However, it is more difficult to use and to learn because the syntax and the documentation are a bit cryptic.

4. The `dratex` package is a small graphic package for creating a graphics. Compared to the other package, including Ti*k*Z, it is very small, which may or may not be an advantage.

5. The `metapost` program is a powerful alternative to Ti*k*Z. It used to be an external program, which entailed a bunch of problems, but in LuaTeX it is now built in. An obstacle with `metapost` is the inclusion of labels. This is *much* easier to achieve using PGF.

6. The `xfig` program is an important alternative to Ti*k*Z for users who do not wish to "program" their graphics as is necessary with Ti*k*Z and the other packages above. There is a conversion program that will convert `xfig` graphics to Ti*k*Z.

1.3 Utility Packages

The PGF package comes along with a number of utility package that are not really about creating graphics and which can be used independently of PGF. However, they are bundled with PGF, partly out of convenience, partly because their functionality is closely intertwined with PGF. These utility packages are:

1. The `pgfkeys` package defines a powerful key management facility. It can be used completely independently of PGF.

2. The `pgffor` package defines a useful `\foreach` statement.

3. The `pgfcalendar` package defines macros for creating calendars. Typically, these calendars will be rendered using PGF's graphic engine, but you can use `pgfcalendar` also typeset calendars using normal text. The package also defines commands for "working" with dates.

4. The `pgfpages` package is used to assemble several pages into a single page. It provides commands for assembling several "virtual pages" into a single "physical page". The idea is that whenever TeX has a page ready for "shipout", `pgfpages` interrupts this shipout and instead stores the page to be shipped out in a special box. When enough "virtual pages" have been accumulated in this way, they are scaled

down and arranged on a "physical page", which then *really* shipped out. This mechanism allows you to create "two page on one page" versions of a document directly inside LaTeX without the use of any external programs. However, `pgfpages` can do quite a lot more than that. You can use it to put logos and watermark on pages, print up to 16 pages on one page, add borders to pages, and more.

1.4 How to Read This Manual

This manual describes both the design of Ti*k*Z and its usage. The organization is very roughly according to "user-friendliness". The commands and subpackages that are easiest and most frequently used are described first, more low-level and esoteric features are discussed later.

If you have not yet installed Ti*k*Z, please read the installation first. Second, it might be a good idea to read the tutorial. Finally, you might wish to skim through the description of Ti*k*Z. Typically, you will not need to read the sections on the basic layer. You will only need to read the part on the system layer if you intend to write your own frontend or if you wish to port PGF to a new driver.

The "public" commands and environments provided by the system are described throughout the text. In each such description, the described command, environment or option is printed in red. Text shown in green is optional and can be left out.

1.5 Authors and Acknowledgements

The bulk of the PGF system and its documentation was written by Till Tantau. A further member of the main team is Mark Wibrow, who is responsible, for example, for the PGF mathematical engine, many shapes, the decoration engine, and matrices. The third member is Christian Feuersänger who contributed the floating point library, image externalization, extended key processing, and automatic hyperlinks in the manual.

Furthermore, occasional contributions have been made by Christophe Jorssen, Jin-Hwan Cho, Olivier Binda, Matthias Schulz, Renée Ahrens, Stephan Schuster, and Thomas Neumann.

Additionally, numerous people have contributed to the PGF system by writing emails, spotting bugs, or sending libraries and patches. Many thanks to all these people, who are too numerous to name them all!

1.6 Getting Help

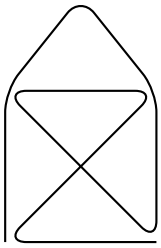
When you need help with PGF and Ti*k*Z, please do the following:

1. Read the manual, at least the part that has to do with your problem.
2. If that does not solve the problem, try having a look at the GitHub development page for PGF and Ti*k*Z (see the title of this document). Perhaps someone has already reported a similar problem and someone has found a solution.
3. On the website you will find numerous forums for getting help. There, you can write to help forums, file bug reports, join mailing lists, and so on.
4. Before you file a bug report, especially a bug report concerning the installation, make sure that this is really a bug. In particular, have a look at the `.log` file that results when you TeX your files. This `.log` file should show that all the right files are loaded from the right directories. Nearly all installation problems can be resolved by looking at the `.log` file.
5. *As a last resort* you can try to email me (Till Tantau) or, if the problem concerns the mathematical engine, Mark Wibrow. I do not mind getting emails, I simply get way too many of them. Because of this, I cannot guarantee that your emails will be answered in a timely fashion or even at all. Your chances that your problem will be fixed are somewhat higher if you mail to the PGF mailing list (naturally, I read this list and answer questions when I have the time).

Part I
Tutorials and Guidelines

by Till Tantau

To help you get started with Ti*k*Z, instead of a long installation and configuration section, this manual starts with tutorials. They explain all the basic and some of the more advanced features of the system, without going into all the details. This part also contains some guidelines on how you should proceed when creating graphics using Ti*k*Z.



```
\tikz \draw[thick,rounded corners=8pt]
  (0,0) -- (0,2) -- (1,3.25) -- (2,2) -- (2,0) -- (0,2) -- (2,2) -- (0,0) -- (2,0);
```

2 Tutorial: A Picture for Karl's Students

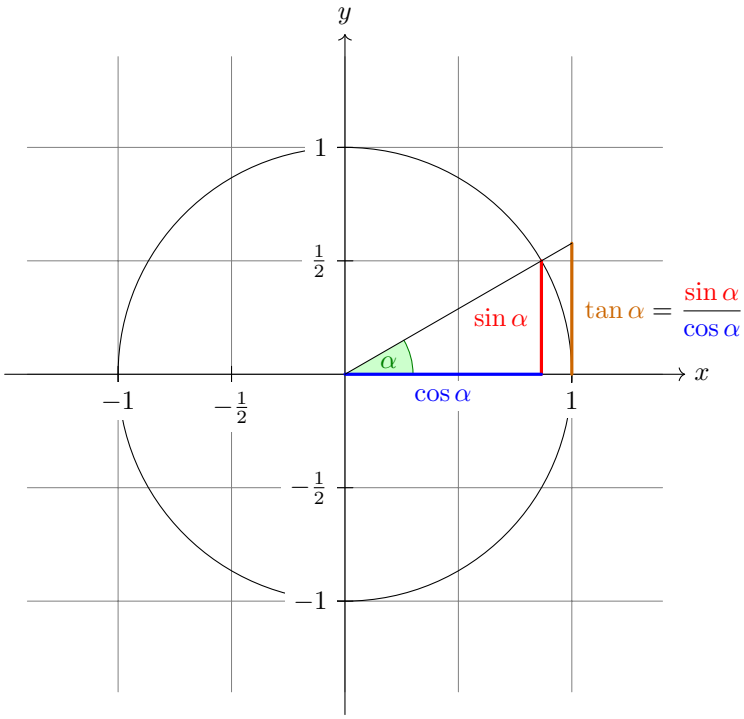
This tutorial is intended for new users of Ti*k*Z. It does not give an exhaustive account of all the features of Ti*k*Z, just of those that you are likely to use right away.

Karl is a math and chemistry high-school teacher. He used to create the graphics in his worksheets and exams using LaTeX's `{picture}` environment. While the results were acceptable, creating the graphics often turned out to be a lengthy process. Also, there tended to be problems with lines having slightly wrong angles and circles also seemed to be hard to get right. Naturally, his students could not care less whether the lines had the exact right angles and they find Karl's exams too difficult no matter how nicely they were drawn. But Karl was never entirely satisfied with the result.

Karl's son, who was even less satisfied with the results (he did not have to take the exams, after all), told Karl that he might wish to try out a new package for creating graphics. A bit confusingly, this package seems to have two names: First, Karl had to download and install a package called PGF. Then it turns out that inside this package there is another package called Ti*k*Z, which is supposed to stand for "Ti*k*Z ist *kein* Zeichenprogramm". Karl finds this all a bit strange and Ti*k*Z seems to indicate that the package does not do what he needs. However, having used GNU software for quite some time and "GNU not being Unix", there seems to be hope yet. His son assures him that Ti*k*Z's name is intended to warn people that Ti*k*Z is not a program that you can use to draw graphics with your mouse or tablet. Rather, it is more like a "graphics language".

2.1 Problem Statement

Karl wants to put a graphic on the next worksheet for his students. He is currently teaching his students about sine and cosine. What he would like to have is something that looks like this (ideally):



The angle α is 30° in the example ($\pi/6$ in radians). The sine of α, which is the height of the red line, is

$$\sin\alpha = 1/2.$$

By the Theorem of Pythagoras we have $\cos^2\alpha + \sin^2\alpha = 1$. Thus the length of the blue line, which is the cosine of α, must be

$$\cos\alpha = \sqrt{1 - 1/4} = \tfrac{1}{2}\sqrt{3}.$$

This shows that $\tan\alpha$, which is the height of the orange line, is

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} = 1/\sqrt{3}.$$

2.2 Setting up the Environment

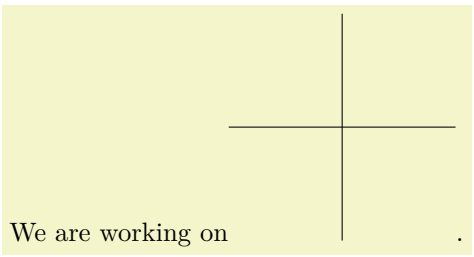
In Ti*k*Z, to draw a picture, at the start of the picture you need to tell TeX or LaTeX that you want to start a picture. In LaTeX this is done using the environment `{tikzpicture}`, in plain TeX you just use `\tikzpicture` to start the picture and `\endtikzpicture` to end it.

2.2.1 Setting up the Environment in LaTeX

Karl, being a LaTeX user, thus sets up his file as follows:

```
\documentclass{article} % say
\usepackage{tikz}
\begin{document}
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
\end{document}
```

When executed, that is, run via `pdflatex` or via `latex` followed by `dvips`, the resulting will contain something that looks like this:




```
We are working on
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\end{tikzpicture}.
```

Admittedly, not quite the whole picture, yet, but we do have the axes established. Well, not quite, but we have the lines that make up the axes drawn. Karl suddenly has a sinking feeling that the picture is still some way off.

Let's have a more detailed look at the code. First, the package `tikz` is loaded. This package is a so-called "frontend" to the basic PGF system. The basic layer, which is also described in this manual, is somewhat more, well, basic and thus harder to use. The frontend makes things easier by providing a simpler syntax.

Inside the environment there are two `\draw` commands. They mean: "The path, which is specified following the command up to the semicolon, should be drawn." The first path is specified as `(-1.5,0) -- (1.5,0)`, which means "a straight line from the point at position $(-1.5, 0)$ to the point at position $(1.5, 0)$". Here, the positions are specified within a special coordinate system in which, initially, one unit is 1cm.

Karl is quite pleased to note that the environment automatically reserves enough space to encompass the picture.

2.2.2 Setting up the Environment in Plain TEX

Karl's wife Gerda, who also happens to be a math teacher, is not a LATEX user, but uses plain TEX since she prefers to do things "the old way". She can also use Ti*k*Z. Instead of `\usepackage{tikz}` she has to write `\input{tikz.tex}` and instead of `\begin{tikzpicture}` she writes `\tikzpicture` and instead of `\end{tikzpicture}` she writes `\endtikzpicture`.

Thus, she would use:

```
%% Plain TeX file
\input{tikz.tex}
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
We are working on
\tikzpicture
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
\endtikzpicture.
\bye
```

Gerda can typeset this file using either `pdftex` or `tex` together with `dvips`. Ti*k*Z will automatically discern which driver she is using. If she wishes to use `dvipdfm` together with `tex`, she either needs to modify the file `pgf.cfg` or can write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` somewhere *before* she inputs `tikz.tex` or `pgf.tex`.

2.2.3 Setting up the Environment in ConTEXt

Karl's uncle Hans uses ConTEXt. Like Gerda, Hans can also use Ti*k*Z. Instead of `\usepackage{tikz}` he says `\usemodule[tikz]`. Instead of `\begin{tikzpicture}` he writes `\starttikzpicture` and instead of `\end{tikzpicture}` he writes `\stoptikzpicture`.

His version of the example looks like this:

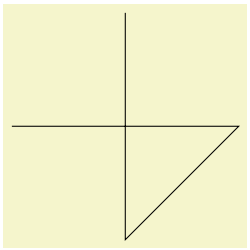
```
%% ConTeXt file
\usemodule[tikz]

\starttext
  We are working on
  \starttikzpicture
    \draw (-1.5,0) -- (1.5,0);
    \draw (0,-1.5) -- (0,1.5);
  \stoptikzpicture.
\stoptext
```

Hans will now typeset this file in the usual way using `texexec` or `context`.

2.3 Straight Path Construction

The basic building block of all pictures in Ti*k*Z is the path. A *path* is a series of straight lines and curves that are connected (that is not the whole picture, but let us ignore the complications for the moment). You start a path by specifying the coordinates of the start position as a point in round brackets, as in `(0,0)`. This is followed by a series of "path extension operations". The simplest is `--`, which we used already. It must be followed by another coordinate and it extends the path in a straight line to this new position. For example, if we were to turn the two paths of the axes into one path, the following would result:



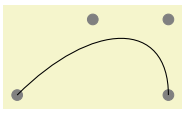
```
\tikz \draw (-1.5,0) -- (1.5,0) -- (0,-1.5) -- (0,1.5);
```

Karl is a bit confused by the fact that there is no `{tikzpicture}` environment, here. Instead, the little command `\tikz` is used. This command either takes one argument (starting with an opening brace as in `\tikz{\draw (0,0) -- (1.5,0)}`, which yields ——————) or collects everything up to the next semicolon and puts it inside a `{tikzpicture}` environment. As a rule of thumb, all Ti*k*Z graphic drawing commands must occur as an argument of `\tikz` or inside a `{tikzpicture}` environment. Fortunately, the command `\draw` will only be defined inside this environment, so there is little chance that you will accidentally do something wrong here.

2.4 Curved Path Construction

The next thing Karl wants to do is to draw the circle. For this, straight lines obviously will not do. Instead, we need some way to draw curves. For this, Ti*k*Z provides a special syntax. One or two "control points" are needed. The math behind them is not quite trivial, but here is the basic idea: Suppose you are at point x and the first control point is y. Then the curve will start "going in the direction of y at x", that is, the tangent of the curve at x will point toward y. Next, suppose the curve should end at z and the second support point is w. Then the curve will, indeed, end at z and the tangent of the curve at point z will go through w.

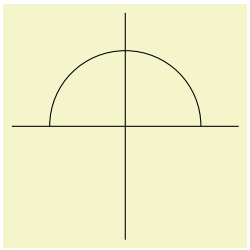
Here is an example (the control points have been added for clarity):



```
\begin{tikzpicture}
  \filldraw [gray] (0,0) circle [radius=2pt]
                   (1,1) circle [radius=2pt]
                   (2,1) circle [radius=2pt]
                   (2,0) circle [radius=2pt];
  \draw (0,0) .. controls (1,1) and (2,1) .. (2,0);
\end{tikzpicture}
```

The general syntax for extending a path in a "curved" way is `.. controls` ⟨*first control point*⟩ `and` ⟨*second control point*⟩ `..` ⟨*end point*⟩. You can leave out the `and` ⟨*second control point*⟩, which causes the first one to be used twice.

So, Karl can now add the first half circle to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (-1,0) .. controls (-1,0.555) and (-0.555,1) .. (0,1)
               .. controls (0.555,1) and (1,0.555) .. (1,0);
\end{tikzpicture}
```

Karl is happy with the result, but finds specifying circles in this way to be extremely awkward. Fortunately, there is a much simpler way.

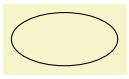
2.5 Circle Path Construction

In order to draw a circle, the path construction operation `circle` can be used. This operation is followed by a radius in brackets as in the following example: (Note that the previous position is used as the *center* of the circle.)



```
\tikz \draw (0,0) circle [radius=10pt];
```

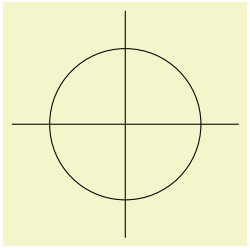
You can also append an ellipse to the path using the `ellipse` operation. Instead of a single radius you can specify two of them:



```
\tikz \draw (0,0) ellipse [x radius=20pt, y radius=10pt];
```

To draw an ellipse whose axes are not horizontal and vertical, but point in an arbitrary direction (a "turned ellipse" like ⬭) you can use transformations, which are explained later. The code for the little ellipse is `\tikz \draw[rotate=30] (0,0) ellipse [x radius=6pt, y radius=3pt];`, by the way.

So, returning to Karl's problem, he can write `\draw (0,0) circle [radius=1cm];` to draw the circle:

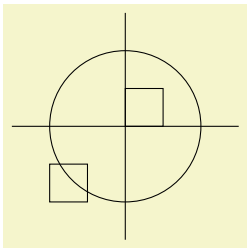


```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

At this point, Karl is a bit alarmed that the circle is so small when he wants the final picture to be much bigger. He is pleased to learn that Ti*k*Z has powerful transformation options and scaling everything by a factor of three is very easy. But let us leave the size as it is for the moment to save some space.

2.6 Rectangle Path Construction

The next things we would like to have is the grid in the background. There are several ways to produce it. For example, one might draw lots of rectangles. Since rectangles are so common, there is a special syntax for them: To add a rectangle to the current path, use the `rectangle` path construction operation. This operation should be followed by another coordinate and will append a rectangle to the path such that the previous coordinate and the next coordinates are corners of the rectangle. So, let us add two rectangles to the picture:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (0,0) rectangle (0.5,0.5);
  \draw (-0.5,-0.5) rectangle (-1,-1);
\end{tikzpicture}
```

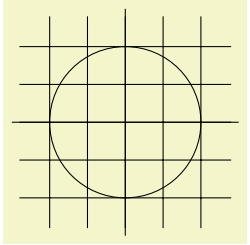
While this may be nice in other situations, this is not really leading anywhere with Karl's problem: First, we would need an awful lot of these rectangles and then there is the border that is not "closed".

So, Karl is about to resort to simply drawing four vertical and four horizontal lines using the nice `\draw` command, when he learns that there is a `grid` path construction operation.

2.7 Grid Path Construction

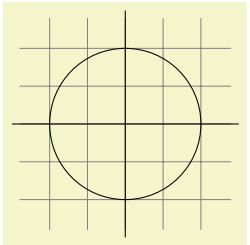
The `grid` path operation adds a grid to the current path. It will add lines making up a grid that fills the rectangle whose one corner is the current point and whose other corner is the point following the `grid` operation. For example, the code `\tikz \draw[step=2pt] (0,0) grid (10pt,10pt);` produces . Note how the optional argument for `\draw` can be used to specify a grid width (there are also `xstep` and `ystep` to define the steppings independently). As Karl will learn soon, there are *lots* of things that can be influenced using such options.

For Karl, the following code could be used:



```
\begin{tikzpicture}
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw[step=.5cm] (-1.4,-1.4) grid (1.4,1.4);
\end{tikzpicture}
```

Having another look at the desired picture, Karl notices that it would be nice for the grid to be more subdued. (His son told him that grids tend to be distracting if they are not subdued.) To subdue the grid, Karl adds two more options to the `\draw` command that draws the grid. First, he uses the color `gray` for the grid lines. Second, he reduces the line width to `very thin`. Finally, he swaps the ordering of the commands so that the grid is drawn first and everything else on top.



```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
\end{tikzpicture}
```

2.8 Adding a Touch of Style

Instead of the options `gray,very thin` Karl could also have said `help lines`. *Styles* are predefined sets of options that can be used to organize how a graphic is drawn. By saying `help lines` you say "use the style that I (or someone else) has set for drawing help lines". If Karl decides, at some later point, that grids should be drawn, say, using the color `blue!50` instead of `gray`, he could provide the following option somewhere:

```
help lines/.style={color=blue!50,very thin}
```

The effect of this "style setter" is that in the current scope or environment the `help lines` option has the same effect as `color=blue!50,very thin`.

Using styles makes your graphics code more flexible. You can change the way things look easily in a consistent manner. Normally, styles are defined at the beginning of a picture. However, you may sometimes wish to define a style globally, so that all pictures of your document can use this style. Then you can easily change the way all graphics look by changing this one style. In this situation you can use the `\tikzset` command at the beginning of the document as in

```
\tikzset{help lines/.style=very thin}
```

To build a hierarchy of styles you can have one style use another. So in order to define a style `Karl's grid` that is based on the `grid` style Karl could say

```
\tikzset{Karl's grid/.style={help lines,color=blue!50}}
...
\draw[Karl's grid] (0,0) grid (5,5);
```

Styles are made even more powerful by parametrization. This means that, like other options, styles can also be used with a parameter. For instance, Karl could parameterize his grid so that, by default, it is blue, but he could also use another color.

```
\begin{tikzpicture}
  [Karl's grid/.style  ={help lines,color=#1!50},
   Karl's grid/.default=blue]

  \draw[Karl's grid]     (0,0) grid (1.5,2);
  \draw[Karl's grid=red] (2,0) grid (3.5,2);
\end{tikzpicture}
```

In this example, the definition of the style `Karl's grid` is given as an optional argument to the `{tikzpicture}` environment. Additional styles for other elements would follow after a comma. With many styles in effect, the optional argument of the environment may easily happen to be longer than the actual contents.

2.9 Drawing Options

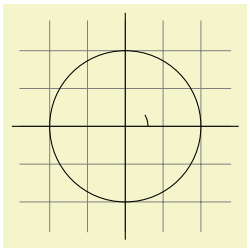
Karl wonders what other options there are that influence how a path is drawn. He saw already that the `color=`⟨*color*⟩ option can be used to set the line's color. The option `draw=`⟨*color*⟩ does nearly the same, only it sets the color for the lines only and a different color can be used for filling (Karl will need this when he fills the arc for the angle).

He saw that the style `very thin` yields very thin lines. Karl is not really surprised by this and neither is he surprised to learn that `thin` yields thin lines, `thick` yields thick lines, `very thick` yields very thick lines, `ultra thick` yields really, really thick lines and `ultra thin` yields lines that are so thin that low-resolution printers and displays will have trouble showing them. He wonders what gives lines of "normal" thickness. It turns out that `thin` is the correct choice, since it gives the same thickness as TEX's `\hrule` command. Nevertheless, Karl would like to know whether there is anything "in the middle" between `thin` and `thick`. There is: `semithick`.

Another useful thing one can do with lines is to dash or dot them. For this, the two styles `dashed` and `dotted` can be used, yielding - - - - and ·····. Both options also exist in a loose and a dense version, called `loosely dashed`, `densely dashed`, `loosely dotted`, and `densely dotted`. If he really, really needs to, Karl can also define much more complex dashing patterns with the `dash pattern` option, but his son insists that dashing is to be used with utmost care and mostly distracts. Karl's son claims that complicated dashing patterns are evil. Karl's students do not care about dashing patterns.

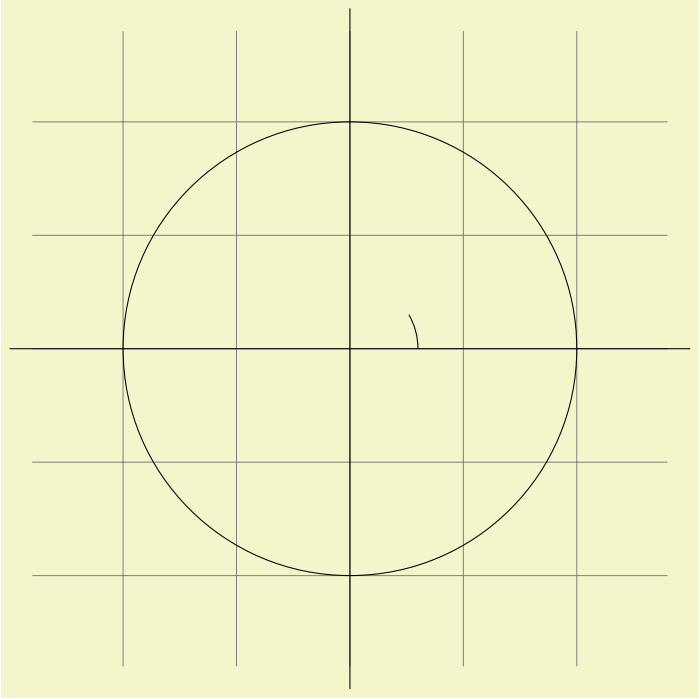
2.10 Arc Path Construction

Our next obstacle is to draw the arc for the angle. For this, the `arc` path construction operation is useful, which draws part of a circle or ellipse. This `arc` operation is followed by options in brackets that specify the arc. An example would be `arc[start angle=10, end angle=80, radius=10pt]`, which means exactly what it says. Karl obviously needs an arc from 0° to 30°. The radius should be something relatively small, perhaps around one third of the circle's radius. When one uses the arc path construction operation, the specified arc will be added with its starting point at the current position. So, we first have to "get there".



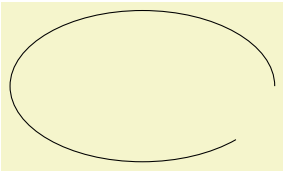
```
\begin{tikzpicture}
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

Karl thinks this is really a bit small and he cannot continue unless he learns how to do scaling. For this, he can add the `[scale=3]` option. He could add this option to each `\draw` command, but that would be awkward. Instead, he adds it to the whole environment, which causes this option to apply to everything within.



```
\begin{tikzpicture}[scale=3]
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

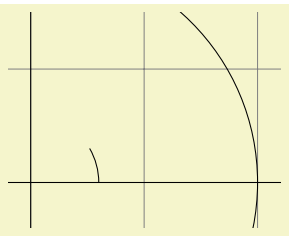
As for circles, you can specify "two" radii in order to get an elliptical arc.



```
\tikz \draw (0,0)
  arc [start angle=0, end angle=315,
       x radius=1.75cm, y radius=1cm];
```

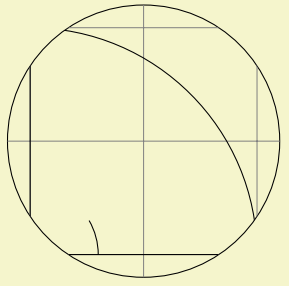
2.11 Clipping a Path

In order to save space in this manual, it would be nice to clip Karl's graphics a bit so that we can focus on the "interesting" parts. Clipping is pretty easy in Ti*k*Z. You can use the `\clip` command to clip all subsequent drawing. It works like `\draw`, only it does not draw anything, but uses the given path to clip everything subsequently.



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

You can also do both at the same time: Draw *and* clip a path. For this, use the `\draw` command and add the `clip` option. (This is not the whole picture: You can also use the `\clip` command and add the `draw` option. Well, that is also not the whole picture: In reality, `\draw` is just a shorthand for `\path[draw]` and `\clip` is a shorthand for `\path[clip]` and you could also say `\path[draw,clip]`.) Here is an example:



```
\begin{tikzpicture}[scale=3]
  \clip[draw] (0.5,0.5) circle [radius=.6cm];
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \draw (3mm,0mm) arc [start angle=0, end angle=30, radius=3mm];
\end{tikzpicture}
```

2.12 Parabola and Sine Path Construction

Although Karl does not need them for his picture, he is pleased to learn that there are `parabola` and `sin` and `cos` path operations for adding parabolas and sine and cosine curves to the current path. For the `parabola` operation, the current point will lie on the parabola as well as the point given after the parabola operation. Consider the following example:



```
\tikz \draw (0,0) rectangle (1,1)  (0,0) parabola (1,1);
```

It is also possible to place the bend somewhere else:



```
\tikz \draw[x=1pt,y=1pt] (0,0) parabola bend (4,16) (6,12);
```

The operations `sin` and `cos` add a sine or cosine curve in the interval $[0, \pi/2]$ such that the previous current point is at the start of the curve and the curve ends at the given end point. Here are two examples:

A sine ⌒ curve.

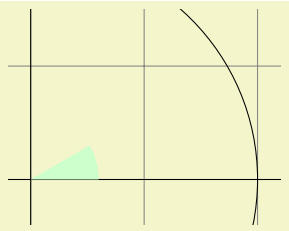
```
A sine \tikz \draw[x=1ex,y=1ex] (0,0) sin (1.57,1); curve.
```



```
\tikz \draw[x=1.57ex,y=1ex] (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
                            (0,1) cos (1,0) sin (2,-1) cos (3,0) sin (4,1);
```

2.13 Filling and Drawing

Returning to the picture, Karl now wants the angle to be "filled" with a very light green. For this he uses `\fill` instead of `\draw`. Here is what Karl does:



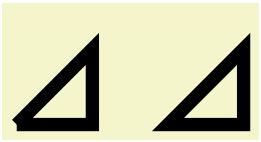
```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \fill[green!20!white] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- (0,0);
\end{tikzpicture}
```

The color `green!20!white` means 20% green and 80% white mixed together. Such color expression are possible since Ti*k*Z uses Uwe Kern's `xcolor` package, see the documentation of that package for details on color expressions.

What would have happened, if Karl had not "closed" the path using `--(0,0)` at the end? In this case, the path is closed automatically, so this could have been omitted. Indeed, it would even have been better to write the following, instead:

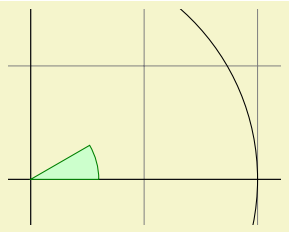
```
\fill[green!20!white] (0,0) -- (3mm,0mm)
  arc [start angle=0, end angle=30, radius=3mm] -- cycle;
```

The `--cycle` causes the current path to be closed (actually the current part of the current path) by smoothly joining the first and last point. To appreciate the difference, consider the following example:



```
\begin{tikzpicture}[line width=5pt]
  \draw (0,0) -- (1,0) -- (1,1) -- (0,0);
  \draw (2,0) -- (3,0) -- (3,1) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

You can also fill and draw a path at the same time using the `\filldraw` command. This will first draw the path, then fill it. This may not seem too useful, but you can specify different colors to be used for filling and for stroking. These are specified as optional arguments like this:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20!white, draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

2.14 Shading

Karl briefly considers the possibility of making the angle "more fancy" by *shading* it. Instead of filling the area with a uniform color, a smooth transition between different colors is used. For this, `\shade` and `\shadedraw`, for shading and drawing at the same time, can be used:



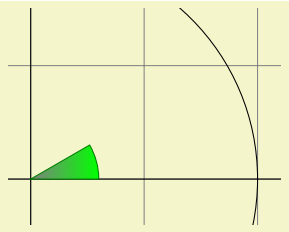
```
\tikz \shade (0,0) rectangle (2,1)  (3,0.5) circle [radius=.5cm];
```

The default shading is a smooth transition from gray to white. To specify different colors, you can use options:



```
\begin{tikzpicture}[rounded corners,ultra thick]
  \shade[top color=yellow,bottom color=black] (0,0) rectangle +(2,1);
  \shade[left color=yellow,right color=black] (3,0) rectangle +(2,1);
  \shadedraw[inner color=yellow,outer color=black,draw=yellow] (6,0) rectangle +(2,1);
  \shade[ball color=green] (9,.5) circle [radius=.5cm];
\end{tikzpicture}
```

For Karl, the following might be appropriate:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \shadedraw[left color=gray,right color=green, draw=green!50!black]
    (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
\end{tikzpicture}
```

However, he wisely decides that shadings usually only distract without adding anything to the picture.

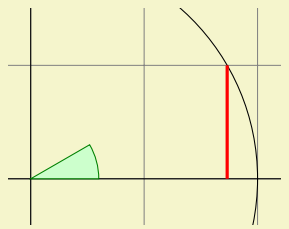
2.15 Specifying Coordinates

Karl now wants to add the sine and cosine lines. He knows already that he can use the `color=` option to set the lines' colors. So, what is the best way to specify the coordinates?

There are different ways of specifying coordinates. The easiest way is to say something like `(10pt,2cm)`. This means 10pt in x-direction and 2cm in y-directions. Alternatively, you can also leave out the units as in `(1,2)`, which means "one times the current x-vector plus twice the current y-vector". These vectors default to 1cm in the x-direction and 1cm in the y-direction, respectively.

In order to specify points in polar coordinates, use the notation `(30:1cm)`, which means 1cm in direction 30 degree. This is obviously quite useful to "get to the point $(\cos 30°, \sin 30°)$ on the circle".

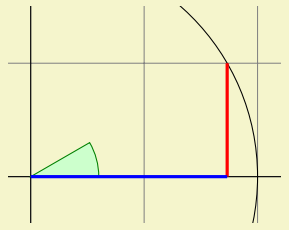
You can add a single `+` sign in front of a coordinate or two of them as in `+(0cm,1cm)` or `++(2cm,0cm)`. Such coordinates are interpreted differently: The first form means "1cm upwards from the previous specified position" and the second means "2cm to the right of the previous specified position, making this the new specified position". For example, we can draw the sine line as follows:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick] (30:1cm) -- +(0,-0.5);
\end{tikzpicture}
```

Karl used the fact $\sin 30° = 1/2$. However, he very much doubts that his students know this, so it would be nice to have a way of specifying "the point straight down from `(30:1cm)` that lies on the x-axis". This is, indeed, possible using a special syntax: Karl can write `(30:1cm |- 0,0)`. In general, the meaning of `(`$\langle p \rangle$ `|-` $\langle q \rangle$`)` is "the intersection of a vertical line through p and a horizontal line through q".

Next, let us draw the cosine line. One way would be to say `(30:1cm |- 0,0) -- (0,0)`. Another way is the following: we "continue" from where the sine ends:

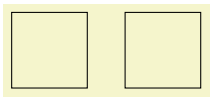


```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,0.75);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw (-1.5,0) -- (1.5,0);
  \draw (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick]  (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick] (30:1cm) ++(0,-0.5) -- (0,0);
\end{tikzpicture}
```

Note that there is no `--` between `(30:1cm)` and `++(0,-0.5)`. In detail, this path is interpreted as follows: "First, the `(30:1cm)` tells me to move my pen to $(\cos 30°, 1/2)$. Next, there comes another coordinate

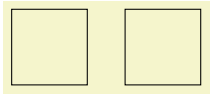
specification, so I move my pen there without drawing anything. This new point is half a unit down from the last position, thus it is at $(\cos 30°, 0)$. Finally, I move the pen to the origin, but this time drawing something (because of the `--`)."

To appreciate the difference between `+` and `++` consider the following example:



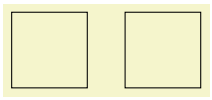
```
\begin{tikzpicture}
  \def\rectanglepath{-- ++(1cm,0cm)  -- ++(0cm,1cm)  -- ++(-1cm,0cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

By comparison, when using a single `+`, the coordinates are different:



```
\begin{tikzpicture}
  \def\rectanglepath{-- +(1cm,0cm)  -- +(1cm,1cm)  -- +(0cm,1cm) -- cycle}
  \draw (0,0) \rectanglepath;
  \draw (1.5,0) \rectanglepath;
\end{tikzpicture}
```

Naturally, all of this could have been written more clearly and more economically like this (either with a single or a double `+`):



```
\tikz \draw (0,0) rectangle +(1,1)  (1.5,0) rectangle +(1,1);
```

2.16 Intersecting Paths

Karl is left with the line for $\tan\alpha$, which seems difficult to specify using transformations and polar coordinates. The first – and easiest – thing he can do is so simply use the coordinate `(1,{tan(30)})` since Ti*k*Z's math engine knows how to compute things like `tan(30)`. Note the added braces since, otherwise, Ti*k*Z's parser would think that the first closing parenthesis ends the coordinate (in general, you need to add braces around components of coordinates when these components contain parentheses).

Karl can, however, also use a more elaborate, but also more "geometric" way of computing the length of the orange line: He can specify intersections of paths as coordinates. The line for $\tan\alpha$ starts at $(1, 0)$ and goes upward to a point that is at the intersection of a line going "up" and a line going from the origin through `(30:1cm)`. Such computations are made available by the `intersections` library.

What Karl must do is to create two "invisible" paths that intersect at the position of interest. Creating paths that are not otherwise seen can be done using the `\path` command without any options like `draw` or `fill`. Then, Karl can add the `name path` option to the path for later reference. Once the paths have been constructed, Karl can use the `name intersections` to assign names to the coordinate for later reference.

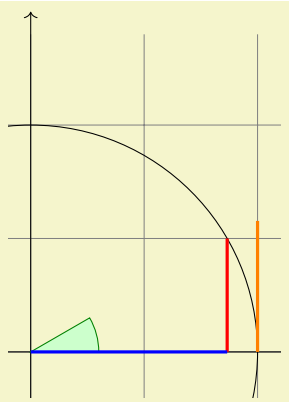
```
\path [name path=upward line] (1,0) -- (1,1);
\path [name path=sloped line] (0,0) -- (30:1.5cm); % a bit longer, so that there is an intersection

% (add `\usetikzlibrary{intersections}' after loading tikz in the preamble)
\draw [name intersections={of=upward line and sloped line, by=x}]
  [very thick,orange] (1,0) -- (x);
```

2.17 Adding Arrow Tips

Karl now wants to add the little arrow tips at the end of the axes. He has noticed that in many plots, even in scientific journals, these arrow tips seem to be missing, presumably because the generating programs cannot produce them. Karl thinks arrow tips belong at the end of axes. His son agrees. His students do not care about arrow tips.

It turns out that adding arrow tips is pretty easy: Karl adds the option `->` to the drawing commands for the axes:

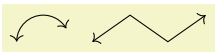


```
\usetikzlibrary {intersections}
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
        arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[red,very thick]    (30:1cm) -- +(0,-0.5);
  \draw[blue,very thick]   (30:1cm) ++(0,-0.5) -- (0,0);

  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=x}]
        [very thick,orange] (1,0) -- (x);
\end{tikzpicture}
```

If Karl had used the option `<-` instead of `->`, arrow tips would have been put at the beginning of the path. The option `<->` puts arrow tips at both ends of the path.

There are certain restrictions to the kind of paths to which arrow tips can be added. As a rule of thumb, you can add arrow tips only to a single open "line". For example, you cannot add tips to, say, a rectangle or a circle. However, you can add arrow tips to curved paths and to paths that have several segments, as in the following examples:



```
\begin{tikzpicture}
  \draw [<->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
  \draw [<->] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl has a more detailed look at the arrow that TikZ puts at the end. It looks like this when he zooms it: →. The shape seems vaguely familiar and, indeed, this is exactly the end of TEX's standard arrow used in something like $f\colon A \to B$.

Karl likes the arrow, especially since it is not "as thick" as the arrows offered by many other packages. However, he expects that, sometimes, he might need to use some other kinds of arrow. To do so, Karl can say `>=`⟨*kind of end arrow tip*⟩, where ⟨*kind of end arrow tip*⟩ is a special arrow tip specification. For example, if Karl says `>=Stealth`, then he tells TikZ that he would like "stealth-fighter-like" arrow tips:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[>=Stealth]
  \draw [->] (0,0) arc [start angle=180, end angle=30, radius=10pt];
  \draw [<<-,very thick] (1,0) -- (1.5cm,10pt) -- (2cm,0pt) -- (2.5cm,10pt);
\end{tikzpicture}
```

Karl wonders whether such a military name for the arrow type is really necessary. He is not really mollified when his son tells him that Microsoft's PowerPoint uses the same name. He decides to have his students discuss this at some point.

In addition to `Stealth` there are several other predefined kinds of arrow tips Karl can choose from, see Section 105. Furthermore, he can define arrows types himself, if he needs new ones.

2.18 Scoping

Karl saw already that there are numerous graphic options that affect how paths are rendered. Often, he would like to apply certain options to a whole set of graphic commands. For example, Karl might wish to draw three paths using a `thick` pen, but would like everything else to be drawn "normally".

If Karl wishes to set a certain graphic option for the whole picture, he can simply pass this option to the `\tikz` command or to the `{tikzpicture}` environment (Gerda would pass the options to `\tikzpicture` and Hans passes them to `\starttikzpicture`). However, if Karl wants to apply graphic options to a local group, he put these commands inside a `{scope}` environment (Gerda uses `\scope` and `\endscope`, Hans uses `\startscope` and `\stopscope`). This environment takes graphic options as an optional argument and these options apply to everything inside the scope, but not to anything outside.

Here is an example:



```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (0,1);
  \begin{scope}[thin]
    \draw (1,0) -- (1,1);
    \draw (2,0) -- (2,1);
  \end{scope}
  \draw (3,0) -- (3,1);
\end{tikzpicture}
```

Scoping has another interesting effect: Any changes to the clipping area are local to the scope. Thus, if you say `\clip` somewhere inside a scope, the effect of the `\clip` command ends at the end of the scope. This is useful since there is no other way of "enlarging" the clipping area.

Karl has also already seen that giving options to commands like `\draw` apply only to that command. It turns out that the situation is slightly more complex. First, options to a command like `\draw` are not really options to the command, but they are "path options" and can be given anywhere on the path. So, instead of `\draw[thin] (0,0) -- (1,0);` one can also write `\draw (0,0) [thin] -- (1,0);` or `\draw (0,0) -- (1,0) [thin];`; all of these have the same effect. This might seem strange since in the last case, it would appear that the `thin` should take effect only "after" the line from $(0,0)$ to $(1,0)$ has been drawn. However, most graphic options only apply to the whole path. Indeed, if you say both `thin` and `thick` on the same path, the last option given will "win".

When reading the above, Karl notices that only "most" graphic options apply to the whole path. Indeed, all transformation options do *not* apply to the whole path, but only to "everything following them on the path". We will have a more detailed look at this in a moment. Nevertheless, all options given during a path construction apply only to this path.

2.19 Transformations

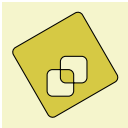
When you specify a coordinate like `(1cm,1cm)`, where is that coordinate placed on the page? To determine the position, Ti*k*Z, TEX, and PDF or PostScript all apply certain transformations to the given coordinate in order to determine the final position on the page.

Ti*k*Z provides numerous options that allow you to transform coordinates in Ti*k*Z's private coordinate system. For example, the `xshift` option allows you to shift all subsequent points by a certain amount:


```
\tikz \draw (0,0) -- (0,0.5) [xshift=2pt] (0,0) -- (0,0.5);
```

It is important to note that you can change transformation "in the middle of a path", a feature that is not supported by PDF or PostScript. The reason is that Ti*k*Z keeps track of its own transformation matrix.

Here is a more complicated example:



```
\begin{tikzpicture}[even odd rule,rounded corners=2pt,x=10pt,y=10pt]
  \filldraw[fill=yellow!80!black] (0,0)   rectangle (1,1)
        [xshift=5pt,yshift=5pt]   (0,0)   rectangle (1,1)
                        [rotate=30]   (-1,-1) rectangle (2,2);
\end{tikzpicture}
```

The most useful transformations are `xshift` and `yshift` for shifting, `shift` for shifting to a given point as in `shift={(1,0)}` or `shift={+(0,0)}` (the braces are necessary so that TEX does not mistake the comma for separating options), `rotate` for rotating by a certain angle (there is also a `rotate around` for rotating around a given point), `scale` for scaling by a certain factor, `xscale` and `yscale` for scaling only in the x- or y-direction (`xscale=-1` is a flip), and `xslant` and `yslant` for slanting. If these transformation and those that I have not mentioned are not sufficient, the `cm` option allows you to apply an arbitrary transformation matrix. Karl's students, by the way, do not know what a transformation matrix is.

2.20 Repeating Things: For-Loops

Karl's next aim is to add little ticks on the axes at positions -1, $-1/2$, $1/2$, and 1. For this, it would be nice to use some kind of "loop", especially since he wishes to do the same thing at each of these positions. There are different packages for doing this. LATEX has its own internal command for this, `pstricks` comes along with the powerful `\multido` command. All of these can be used together with Ti*k*Z, so if you are familiar with them, feel free to use them. Ti*k*Z introduces yet another command, called `\foreach`, which

I introduced since I could never remember the syntax of the other packages. `\foreach` is defined in the package `pgffor` and can be used independently of Ti*k*Z, but Ti*k*Z includes it automatically.

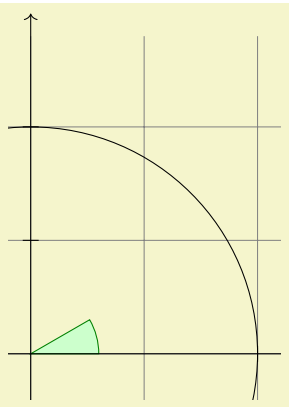
In its basic form, the `\foreach` command is easy to use:

$x = 1$, $x = 2$, $x = 3$,

```
\foreach \x in {1,2,3} {$x =\x$, }
```

The general syntax is `\foreach` ⟨*variable*⟩ `in {`⟨*list of values*⟩`}` ⟨*commands*⟩. Inside the ⟨*commands*⟩, the ⟨*variable*⟩ will be assigned to the different values. If the ⟨*commands*⟩ do not start with a brace, everything up to the next semicolon is used as ⟨*commands*⟩.

For Karl and the ticks on the axes, he could use the following code:



```
\begin{tikzpicture}[scale=3]
  \clip (-0.1,-0.2) rectangle (1.1,1.51);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);
  \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x in {-1cm,-0.5cm,1cm}
    \draw (\x,-1pt) -- (\x,1pt);
  \foreach \y in {-1cm,-0.5cm,0.5cm,1cm}
    \draw (-1pt,\y) -- (1pt,\y);
\end{tikzpicture}
```

As a matter of fact, there are many different ways of creating the ticks. For example, Karl could have put the `\draw ...;` inside curly braces. He could also have used, say,

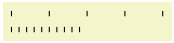
```
\foreach \x in {-1,-0.5,1}
  \draw[xshift=\x cm] (0pt,-1pt) -- (0pt,1pt);
```

Karl is curious what would happen in a more complicated situation where there are, say, 20 ticks. It seems bothersome to explicitly mention all these numbers in the set for `\foreach`. Indeed, it is possible to use `...` inside the `\foreach` statement to iterate over a large number of values (which must, however, be dimensionless real numbers) as in the following example:



```
\tikz \foreach \x in {1,...,10}
        \draw (\x,0) circle [radius=0.4cm];
```

If you provide *two* numbers before the `...`, the `\foreach` statement will use their difference for the stepping, but be wary of floating-point rounding errors:



```
\begin{tikzpicture}
  \foreach \x in {-1,-0.5,...,1}
    \draw (\x cm,-1pt) -- (\x cm,1pt);
  \foreach \x in {-1,-0.9,...,0}
    % tick at 0 is not drawn due to rounding errors
    \draw (\x cm,-7pt) -- (\x cm,-5pt);
\end{tikzpicture}
```

We can also nest loops to create interesting effects:

1,5	2,5	3,5	4,5	5,5		7,5	8,5	9,5	10,5	11,5	12,5
1,4	2,4	3,4	4,4	5,4		7,4	8,4	9,4	10,4	11,4	12,4
1,3	2,3	3,3	4,3	5,3		7,3	8,3	9,3	10,3	11,3	12,3
1,2	2,2	3,2	4,2	5,2		7,2	8,2	9,2	10,2	11,2	12,2
1,1	2,1	3,1	4,1	5,1		7,1	8,1	9,1	10,1	11,1	12,1

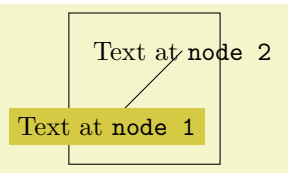
```
\begin{tikzpicture}
  \foreach \x in {1,2,...,5,7,8,...,12}
    \foreach \y in {1,...,5}
    {
      \draw (\x,\y) +(-.5,-.5) rectangle ++(.5,.5);
      \draw (\x,\y) node{\x,\y};
    }
\end{tikzpicture}
```

The `\foreach` statement can do even trickier stuff, but the above gives the idea.

2.21 Adding Text

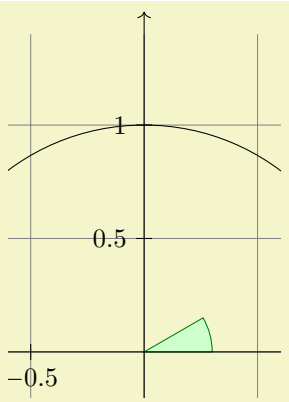
Karl is, by now, quite satisfied with the picture. However, the most important parts, namely the labels, are still missing!

Ti*k*Z offers an easy-to-use and powerful system for adding text and, more generally, complex shapes to a picture at specific positions. The basic idea is the following: When Ti*k*Z is constructing a path and encounters the keyword `node` in the middle of a path, it reads a *node specification*. The keyword `node` is typically followed by some options and then some text between curly braces. This text is put inside a normal TEX box (if the node specification directly follows a coordinate, which is usually the case, Ti*k*Z is able to perform some magic so that it is even possible to use verbatim text inside the boxes) and then placed at the current position, that is, at the last specified position (possibly shifted a bit, according to the given options). However, all nodes are drawn only after the path has been completely drawn/filled/shaded/clipped/whatever.




```
\begin{tikzpicture}
  \draw (0,0) rectangle (2,2);
  \draw (0.5,0.5) node [fill=yellow!80!black]
                      {Text at \verb!node 1!}
     -- (1.5,1.5) node {Text at \verb!node 2!};
\end{tikzpicture}
```

Obviously, Karl would not only like to place nodes *on* the last specified position, but also to the left or the right of these positions. For this, every node object that you put in your picture is equipped with several *anchors*. For example, the `north` anchor is in the middle at the upper end of the shape, the `south` anchor is at the bottom and the `north east` anchor is in the upper right corner. When you give the option `anchor=north`, the text will be placed such that this northern anchor will lie on the current position and the text is, thus, below the current position. Karl uses this to draw the ticks as follows:



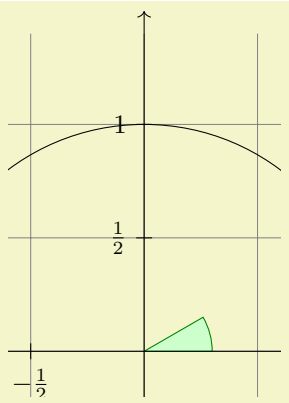

```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0);   \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x in {-1,-0.5,1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\x$};
  \foreach \y in {-1,-0.5,0.5,1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\y$};
\end{tikzpicture}
```

This is quite nice, already. Using these anchors, Karl can now add most of the other text elements. However, Karl thinks that, though "correct", it is quite counter-intuitive that in order to place something *below* a given point, he has to use the *north* anchor. For this reason, there is an option called `below`, which does the same as `anchor=north`. Similarly, `above right` does the same as `anchor=south west`. In addition, `below` takes an optional dimension argument. If given, the shape will additionally be shifted downwards by the given amount. So, `below=1pt` can be used to put a text label below some point and, additionally shift it 1pt downwards.

Karl is not quite satisfied with the ticks. He would like to have 1/2 or $\frac{1}{2}$ shown instead of 0.5, partly to show off the nice capabilities of TEX and Ti*k*Z, partly because for positions like 1/3 or π it is certainly very much preferable to have the "mathematical" tick there instead of just the "numeric" tick. His students, on the other hand, prefer 0.5 over 1/2 since they are not too fond of fractions in general.

Karl now faces a problem: For the `\foreach` statement, the position `\x` should still be given as `0.5` since Ti*k*Z will not know where `\frac{1}{2}` is supposed to be. On the other hand, the typeset text should really be `\frac{1}{2}`. To solve this problem, `\foreach` offers a special syntax: Instead of having one variable `\x`, Karl can specify two (or even more) variables separated by a slash as in `\x / \xtext`. Then, the elements in the set over which `\foreach` iterates must also be of the form ⟨*first*⟩/⟨*second*⟩. In each iteration, `\x` will be set to ⟨*first*⟩ and `\xtext` will be set to ⟨*second*⟩. If no ⟨*second*⟩ is given, the ⟨*first*⟩ will be used again. So, here is the new code for the ticks:

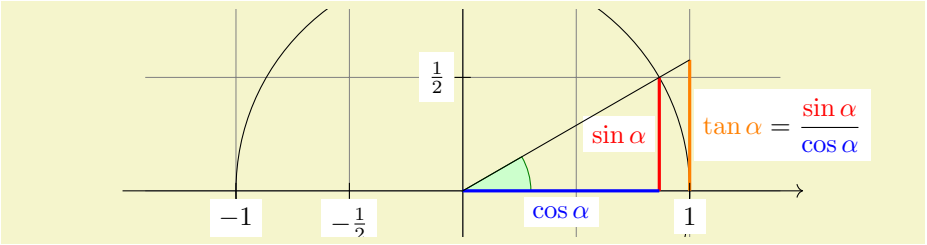



```
\begin{tikzpicture}[scale=3]
  \clip (-0.6,-0.2) rectangle (0.6,1.51);
  \draw[step=.5cm,help lines] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
      arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0); \draw[->] (0,-1.5) -- (0,1.5);
  \draw (0,0) circle [radius=1cm];

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east] {$\ytext$};
\end{tikzpicture}
```

Karl is quite pleased with the result, but his son points out that this is still not perfectly satisfactory: The grid and the circle interfere with the numbers and decrease their legibility. Karl is not very concerned by this (his students do not even notice), but his son insists that there is an easy solution: Karl can add the `[fill=white]` option to fill out the background of the text shape with a white color.

The next thing Karl wants to do is to add the labels like $\sin\alpha$. For this, he would like to place a label "in the middle of the line". To do so, instead of specifying the label `node {$\sin\alpha$}` directly after one of the endpoints of the line (which would place the label at that endpoint), Karl can give the label directly after the `--`, before the coordinate. By default, this places the label in the middle of the line, but the `pos=` options can be used to modify this. Also, options like `near start` and `near end` can be used to modify this position:



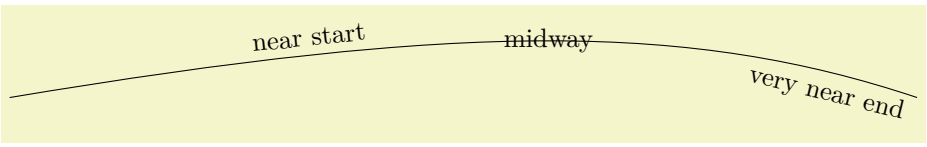

```
\usetikzlibrary {intersections}
\begin{tikzpicture}[scale=3]
  \clip (-2,-0.2) rectangle (2,0.8);
  \draw[step=.5cm,gray,very thin] (-1.4,-1.4) grid (1.4,1.4);
  \filldraw[fill=green!20,draw=green!50!black] (0,0) -- (3mm,0mm)
    arc [start angle=0, end angle=30, radius=3mm] -- cycle;
  \draw[->] (-1.5,0) -- (1.5,0) coordinate (x axis);
  \draw[->] (0,-1.5) -- (0,1.5) coordinate (y axis);
  \draw (0,0) circle [radius=1cm];

  \draw[very thick,red]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);
  \draw[very thick,blue]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);
  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=t}]
    [very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);

  \draw (0,0) -- (t);

  \foreach \x/\xtext in {-1, -0.5/-\frac{1}{2}, 1}
    \draw (\x cm,1pt) -- (\x cm,-1pt) node[anchor=north,fill=white] {$\xtext$};
  \foreach \y/\ytext in {-1, -0.5/-\frac{1}{2}, 0.5/\frac{1}{2}, 1}
    \draw (1pt,\y cm) -- (-1pt,\y cm) node[anchor=east,fill=white] {$\ytext$};
\end{tikzpicture}
```

You can also position labels on curves and, by adding the `sloped` option, have them rotated such that they match the line's slope. Here is an example:




```
\begin{tikzpicture}
  \draw (0,0) .. controls (6,1) and (9,1) ..
    node[near start,sloped,above] {near start}
    node {midway}
    node[very near end,sloped,below] {very near end} (12,0);
\end{tikzpicture}
```

It remains to draw the explanatory text at the right of the picture. The main difficulty here lies in limiting the width of the text "label", which is quite long, so that line breaking is used. Fortunately, Karl can use the option `text width=6cm` to get the desired effect. So, here is the full code:

```
\begin{tikzpicture}
  [scale=3,line cap=round,
  % Styles
  axes/.style=,
  important line/.style={very thick},
  information text/.style={rounded corners,fill=red!10,inner sep=1ex}]

  % Colors
  \colorlet{anglecolor}{green!50!black}
  \colorlet{sincolor}{red}
  \colorlet{tancolor}{orange!80!black}
  \colorlet{coscolor}{blue}

  % The graphic
  \draw[help lines,step=0.5cm] (-1.4,-1.4) grid (1.4,1.4);

  \draw (0,0) circle [radius=1cm];

  \begin{scope}[axes]
    \draw[->] (-1.5,0) -- (1.5,0) node[right] {$x$} coordinate(x axis);
    \draw[->] (0,-1.5) -- (0,1.5) node[above] {$y$} coordinate(y axis);

    \foreach \x/\xtext in {-1, -.5/-\frac{1}{2}, 1}
      \draw[xshift=\x cm] (0pt,1pt) -- (0pt,-1pt) node[below,fill=white] {$\xtext$};

    \foreach \y/\ytext in {-1, -.5/-\frac{1}{2}, .5/\frac{1}{2}, 1}
      \draw[yshift=\y cm] (1pt,0pt) -- (-1pt,0pt) node[left,fill=white] {$\ytext$};
  \end{scope}

  \filldraw[fill=green!20,draw=anglecolor] (0,0) -- (3mm,0pt)
    arc [start angle=0, end angle=30, radius=3mm];
  \draw (15:2mm) node[anglecolor] {$\alpha$};

  \draw[important line,sincolor]
    (30:1cm) -- node[left=1pt,fill=white] {$\sin \alpha$} (30:1cm |- x axis);

  \draw[important line,coscolor]
    (30:1cm |- x axis) -- node[below=2pt,fill=white] {$\cos \alpha$} (0,0);

  \path [name path=upward line] (1,0) -- (1,1);
  \path [name path=sloped line] (0,0) -- (30:1.5cm);
  \draw [name intersections={of=upward line and sloped line, by=t}]
    [very thick,orange] (1,0) -- node [right=1pt,fill=white]
    {$\displaystyle \tan \alpha \color{black}=
      \frac{{\color{red}\sin \alpha}}{\color{blue}\cos \alpha}$} (t);

  \draw (0,0) -- (t);

  \draw[xshift=1.85cm]
    node[right,text width=6cm,information text]
    {
      The {\color{anglecolor} angle $\alpha$} is $30^\circ$ in the
      example ($\pi/6$ in radians). The {\color{sincolor}sine of
        $\alpha$}, which is the height of the red line, is
      \[
      {\color{sincolor} \sin \alpha} = 1/2.
      \]
      By the Theorem of Pythagoras ...
    };
\end{tikzpicture}
```

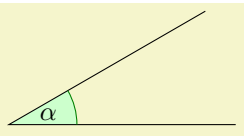
2.22 Pics: The Angle Revisited

Karl expects that the code of certain parts of the picture he created might be so useful that he might wish to reuse them in the future. A natural thing to do is to create TEX macros that store the code he wishes to reuse. However, Ti*k*Z offers another way that is integrated directly into its parser: pics!

A "pic" is "not quite a full picture", hence the short name. The idea is that a pic is simply some code that you can add to a picture at different places using the `pic` command whose syntax is almost identical to the `node` command. The main difference is that instead of specifying some text in curly braces that should be shown, you specify the name of a predefined picture that should be shown.

Defining new pics is easy enough, see Section 18, but right now we just want to use one such predefined pic: the `angle` pic. As the name suggests, it is a small drawing of an angle consisting of a little wedge and an arc together with some text (Karl needs to load the `angles` library and the `quotes` for the following examples). What makes this pic useful is the fact that the size of the wedge will be computed automatically.

The `angle` pic draws an angle between the two lines BA and BC, where A, B, and C are three coordinates. In our case, B is the origin, A is somewhere on the x-axis and C is somewhere on a line at 30°.




```
\usetikzlibrary {angles,quotes}
\begin{tikzpicture}[scale=3]
  \coordinate (A) at (1,0);
  \coordinate (B) at (0,0);
  \coordinate (C) at (30:1cm);

  \draw (A) -- (B) -- (C)
        pic [draw=green!50!black, fill=green!20, angle radius=9mm,
             "$\alpha$"] {angle = A--B--C};
\end{tikzpicture}
```

Let us see, what is happening here. First we have specified three *coordinates* using the `\coordinate` command. It allows us to name a specific coordinate in the picture. Then comes something that starts as a normal `\draw`, but then comes the `pic` command. This command gets lots of options and, in curly braces, comes the most important point: We specify that we want to add an `angle` pic and this angle should be between the points we named `A`, `B`, and `C` (we could use other names). Note that the text that we want to be shown in the pic is specified in quotes inside the options of the `pic`, not inside the curly braces.

To learn more about pics, please see Section 18.

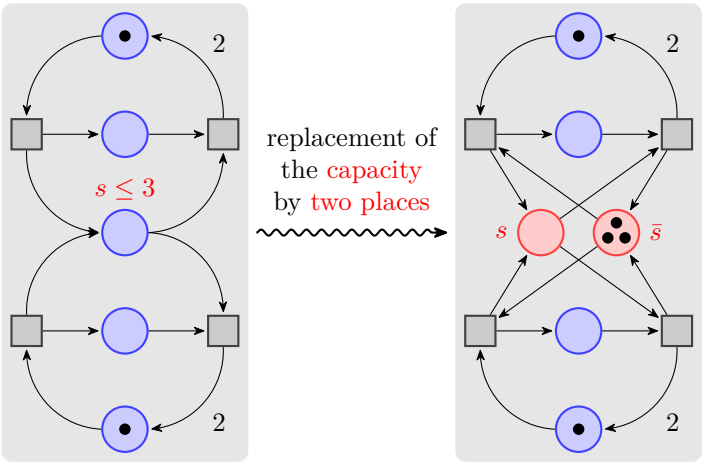
3 Tutorial: A Petri-Net for Hagen

In this second tutorial we explore the node mechanism of Ti*k*Z and PGF.

Hagen must give a talk tomorrow about his favorite formalism for distributed systems: Petri nets! Hagen used to give his talks using a blackboard and everyone seemed to be perfectly content with this. Unfortunately, his audience has been spoiled recently with fancy projector-based presentations and there seems to be a certain amount of peer pressure that his Petri nets should also be drawn using a graphic program. One of the professors at his institute recommends Ti*k*Z for this and Hagen decides to give it a try.

3.1 Problem Statement

For his talk, Hagen wishes to create a graphic that demonstrates how a net with place capacities can be simulated by a net without capacities. The graphic should look like this, ideally:



3.2 Setting up the Environment

For the picture Hagen will need to load the Ti*k*Z package as did Karl in the previous tutorial. However, Hagen will also need to load some additional *library packages* that Karl did not need. These library packages contain additional definitions like extra arrow tips that are typically not needed in a picture and that need to be loaded explicitly.

Hagen will need to load several libraries: The `arrows.meta` library for the special arrow tip used in the graphic, the `decorations.pathmorphing` library for the "snaking line" in the middle, the `backgrounds` library for the two rectangular areas that are behind the two main parts of the picture, the `fit` library to easily compute the sizes of these rectangles, and the `positioning` library for placing nodes relative to other nodes.

3.2.1 Setting up the Environment in LaTeX

When using LaTeX use:

```
\documentclass{article} % say

\usepackage{tikz}
\usetikzlibrary{arrows.meta,decorations.pathmorphing,backgrounds,positioning,fit,petri}

\begin{document}
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
\end{tikzpicture}
\end{document}
```

3.2.2 Setting up the Environment in Plain TeX

When using plain TeX use:

```
%% Plain TeX file
\input{tikz.tex}
\usetikzlibrary{arrows.meta,decorations.pathmorphing,backgrounds,positioning,fit,petri}
\baselineskip=12pt
\hsize=6.3truein
\vsize=8.7truein
\tikzpicture
  \draw (0,0) -- (1,1);
\endtikzpicture
\bye
```

3.2.3 Setting up the Environment in ConTEXt

When using ConTEXt, use:

```
%% ConTeXt file
\usemodule[tikz]
\usetikzlibrary[arrows.meta,decorations.pathmorphing,backgrounds,positioning,fit,petri]

\starttext
  \starttikzpicture
    \draw (0,0) -- (1,1);
  \stoptikzpicture
\stoptext
```

3.3 Introduction to Nodes

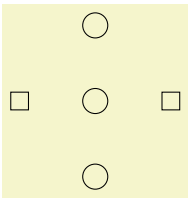
In principle, we already know how to create the graphics that Hagen desires (except perhaps for the snaked line, we will come to that): We start with big light gray rectangle and then add lots of circles and small rectangle, plus some arrows.

However, this approach has numerous disadvantages: First, it is hard to change anything at a later stage. For example, if we decide to add more places to the Petri nets (the circles are called places in Petri net theory), all of the coordinates change and we need to recalculate everything. Second, it is hard to read the code for the Petri net as it is just a long and complicated list of coordinates and drawing commands – the underlying structure of the Petri net is lost.

Fortunately, Ti*k*Z offers a powerful mechanism for avoiding the above problems: nodes. We already came across nodes in the previous tutorial, where we used them to add labels to Karl's graphic. In the present tutorial we will see that nodes are much more powerful.

A node is a small part of a picture. When a node is created, you provide a position where the node should be drawn and a *shape*. A node of shape `circle` will be drawn as a circle, a node of shape `rectangle` as a rectangle, and so on. A node may also contain some text, which is why Karl used nodes to show text. Finally, a node can get a *name* for later reference.

In Hagen's picture we will use nodes for the places and for the transitions of the Petri net (the places are the circles, the transitions are the rectangles). Let us start with the upper half of the left Petri net. In this upper half we have three places and two transitions. Instead of drawing three circles and two rectangles, we use three nodes of shape `circle` and two nodes of shape `rectangle`.



```
\begin{tikzpicture}
  \path ( 0,2) node [shape=circle,draw] {}
        ( 0,1) node [shape=circle,draw] {}
        ( 0,0) node [shape=circle,draw] {}
        ( 1,1) node [shape=rectangle,draw] {}
        (-1,1) node [shape=rectangle,draw] {};
\end{tikzpicture}
```

Hagen notes that this does not quite look like the final picture, but it seems like a good first step.

Let us have a more detailed look at the code. The whole picture consists of a single path. Ignoring the `node` operations, there is not much going on in this path: It is just a sequence of coordinates with nothing "happening" between them. Indeed, even if something were to happen like a line-to or a curve-to, the `\path` command would not "do" anything with the resulting path. So, all the magic must be in the `node` commands.

In the previous tutorial we learned that a `node` will add a piece of text at the last coordinate. Thus, each of the five nodes is added at a different position. In the above code, this text is empty (because of the

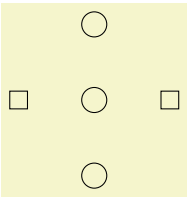
empty {}). So, why do we see anything at all? The answer is the draw option for the node operation: It causes the "shape around the text" to be drawn.

So, the code (0,2) node [shape=circle,draw] {} means the following: "In the main path, add a move-to to the coordinate (0,2). Then, temporarily suspend the construction of the main path while the node is built. This node will be a circle around an empty text. This circle is to be drawn, but not filled or otherwise used. Once this whole node is constructed, it is saved until after the main path is finished. Then, it is drawn." The following (0,1) node [shape=circle,draw] {} then has the following effect: "Continue the main path with a move-to to (0,1). Then construct a node at this position also. This node is also shown after the main path is finished." And so on.

3.4 Placing Nodes Using the At Syntax

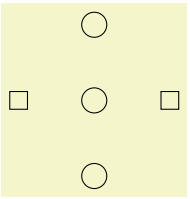
Hagen now understands how the node operation adds nodes to the path, but it seems a bit silly to create a path using the \path operation, consisting of numerous superfluous move-to operations, only to place nodes. He is pleased to learn that there are ways to add nodes in a more sensible manner.

First, the node operation allows one to add at (⟨*coordinate*⟩) in order to directly specify where the node should be placed, sidestepping the rule that nodes are placed on the last coordinate. Hagen can then write the following:



```
\begin{tikzpicture}
  \path node at ( 0,2) [shape=circle,draw] {}
        node at ( 0,1) [shape=circle,draw] {}
        node at ( 0,0) [shape=circle,draw] {}
        node at ( 1,1) [shape=rectangle,draw] {}
        node at (-1,1) [shape=rectangle,draw] {};
\end{tikzpicture}
```

Now Hagen is still left with a single empty path, but at least the path no longer contains strange move-to's. It turns out that this can be improved further: The \node command is an abbreviation for \path node, which allows Hagen to write:

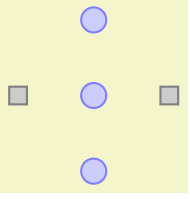


```
\begin{tikzpicture}
  \node at ( 0,2) [circle,draw] {};
  \node at ( 0,1) [circle,draw] {};
  \node at ( 0,0) [circle,draw] {};
  \node at ( 1,1) [rectangle,draw] {};
  \node at (-1,1) [rectangle,draw] {};
\end{tikzpicture}
```

Hagen likes this syntax much better than the previous one. Note that Hagen has also omitted the shape= since, like color=, Ti*k*Z allows you to omit the shape= if there is no confusion.

3.5 Using Styles

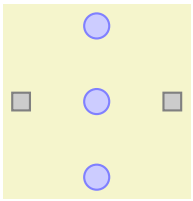
Feeling adventurous, Hagen tries to make the nodes look nicer. In the final picture, the circles and rectangle should be filled with different colors, resulting in the following code:



```
\begin{tikzpicture}[thick]
  \node at ( 0,2) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,1) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 0,0) [circle,draw=blue!50,fill=blue!20] {};
  \node at ( 1,1) [rectangle,draw=black!50,fill=black!20] {};
  \node at (-1,1) [rectangle,draw=black!50,fill=black!20] {};
\end{tikzpicture}
```

While this looks nicer in the picture, the code starts to get a bit ugly. Ideally, we would like our code to transport the message "there are three places and two transitions" and not so much which filling colors should be used.

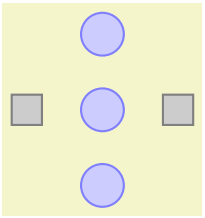
To solve this problem, Hagen uses styles. He defines a style for places and another style for transitions:



```
\begin{tikzpicture}
  [place/.style={circle,draw=blue!50,fill=blue!20,thick},
   transition/.style={rectangle,draw=black!50,fill=black!20,thick}]
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.6 Node Size

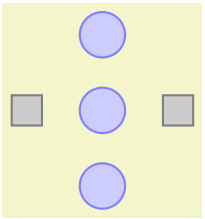
Before Hagen starts naming and connecting the nodes, let us first make sure that the nodes get their final appearance. They are still too small. Indeed, Hagen wonders why they have any size at all, after all, the text is empty. The reason is that Ti*k*Z automatically adds some space around the text. The amount is set using the option `inner sep`. So, to increase the size of the nodes, Hagen could write:



```
\begin{tikzpicture}
  [inner sep=2mm,
   place/.style={circle,draw=blue!50,fill=blue!20,thick},
   transition/.style={rectangle,draw=black!50,fill=black!20,thick}]
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

However, this is not really the best way to achieve the desired effect. It is much better to use the `minimum size` option instead. This option allows Hagen to specify a minimum size that the node should have. If the node actually needs to be bigger because of a longer text, it will be larger, but if the text is empty, then the node will have `minimum size`. This option is also useful to ensure that several nodes containing different amounts of text have the same size. The options `minimum height` and `minimum width` allow you to specify the minimum height and width independently.

So, what Hagen needs to do is to provide `minimum size` for the nodes. To be on the safe side, he also sets `inner sep=0pt`. This ensures that the nodes will really have size `minimum size` and not, for very small minimum sizes, the minimal size necessary to encompass the automatically added space.



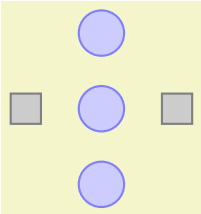
```
\begin{tikzpicture}
  [place/.style={circle,draw=blue!50,fill=blue!20,thick,
                 inner sep=0pt,minimum size=6mm},
   transition/.style={rectangle,draw=black!50,fill=black!20,thick,
                      inner sep=0pt,minimum size=4mm}]
  \node at ( 0,2) [place] {};
  \node at ( 0,1) [place] {};
  \node at ( 0,0) [place] {};
  \node at ( 1,1) [transition] {};
  \node at (-1,1) [transition] {};
\end{tikzpicture}
```

3.7 Naming Nodes

Hagen's next aim is to connect the nodes using arrows. This seems like a tricky business since the arrows should not start in the middle of the nodes, but somewhere on the border and Hagen would very much like to avoid computing these positions by hand.

Fortunately, PGF will perform all the necessary calculations for him. However, he first has to assign names to the nodes so that he can reference them later on.

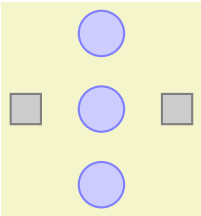
There are two ways to name a node. The first is to use the `name=` option. The second method is to write the desired name in parentheses after the `node` operation. Hagen thinks that this second method seems strange, but he will soon change his opinion.



```
% ... set up styles
\begin{tikzpicture}
  \node (waiting 1)      at ( 0,2) [place] {};
  \node (critical 1)     at ( 0,1) [place] {};
  \node (semaphore)      at ( 0,0) [place] {};
  \node (leave critical) at ( 1,1) [transition] {};
  \node (enter critical) at (-1,1) [transition] {};
\end{tikzpicture}
```

Hagen is pleased to note that the names help in understanding the code. Names for nodes can be pretty arbitrary, but they should not contain commas, periods, parentheses, colons, and some other special characters. However, they can contain underscores and hyphens.

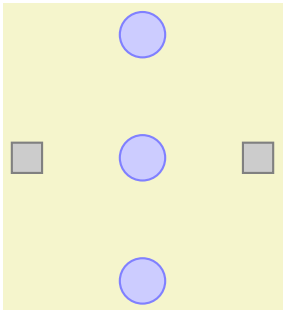
The syntax for the `node` operation is quite liberal with respect to the order in which node names, the `at` specifier, and the options must come. Indeed, you can even have multiple option blocks between the `node` and the text in curly braces, they accumulate. You can rearrange them arbitrarily and perhaps the following might be preferable:



```
\begin{tikzpicture}
  \node[place]      (waiting 1)      at ( 0,2) {};
  \node[place]      (critical 1)     at ( 0,1) {};
  \node[place]      (semaphore)      at ( 0,0) {};
  \node[transition] (leave critical) at ( 1,1) {};
  \node[transition] (enter critical) at (-1,1) {};
\end{tikzpicture}
```

3.8 Placing Nodes Using Relative Placement

Although Hagen still wishes to connect the nodes, he first wishes to address another problem again: The placement of the nodes. Although he likes the `at` syntax, in this particular case he would prefer placing the nodes "relative to each other". So, Hagen would like to say that the `critical 1` node should be below the `waiting 1` node, wherever the `waiting 1` node might be. There are different ways of achieving this, but the nicest one in Hagen's case is the `below` option:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
\end{tikzpicture}
```

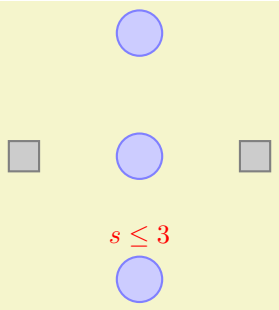
With the `positioning` library loaded, when an option like `below` is followed by `of`, then the position of the node is shifted in such a manner that it is placed at the distance `node distance` in the specified direction of the given direction. The `node distance` is either the distance between the centers of the nodes (when the `on grid` option is set to true) or the distance between the borders (when the `on grid` option is set to false, which is the default).

Even though the above code has the same effect as the earlier code, Hagen can pass it to his colleagues who will be able to just read and understand it, perhaps without even having to see the picture.

3.9 Adding Labels Next to Nodes

Before we have a look at how Hagen can connect the nodes, let us add the capacity "$s \leq 3$" to the bottom node. For this, two approaches are possible:

1. Hagen can just add a new node above the `north` anchor of the `semaphore` node.

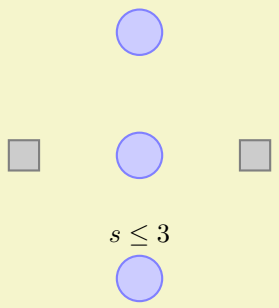



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]  {};

  \node [red,above] at (semaphore.north) {$s\le 3$};
\end{tikzpicture}
```

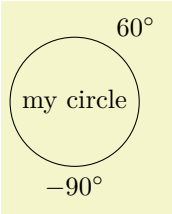
This is a general approach that will "always work".

2. Hagen can use the special `label` option. This option is given to a `node` and it causes *another* node to be added next to the node where the option is given. Here is the idea: When we construct the `semaphore` node, we wish to indicate that we want another node with the capacity above it. For this, we use the option `label=above:$s\le 3$`. This option is interpreted as follows: We want a node above the `semaphore` node and this node should read "$s \le 3$". Instead of `above` we could also use things like `below left` before the colon or a number like `60`.



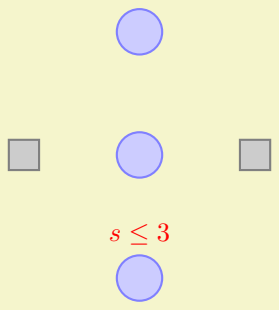

```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]  {};
\end{tikzpicture}
```

It is also possible to give multiple `label` options, this causes multiple labels to be drawn.




```
\tikz
  \node [circle,draw,label=60:$60^\circ$,label=below:$-90^\circ$] {my circle};
```

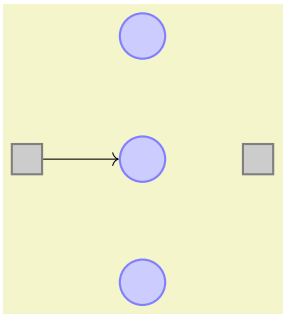
Hagen is not fully satisfied with the `label` option since the label is not red. To achieve this, he has two options: First, he can redefine the `every label` style. Second, he can add options to the label's node. These options are given following the `label=`, so he would write `label=[red]above:$s\le3$`. However, this does not quite work since TEX thinks that the `]` closes the whole option list of the `semaphore` node. So, Hagen has to add braces and writes `label={[red]above:$s\le3$}`. Since this looks a bit ugly, Hagen decides to redefine the `every label` style.




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every label/.style={red}]
  \node[place]      (waiting)                          {};
  \node[place]      (critical)       [below=of waiting]  {};
  \node[place]      (semaphore)      [below=of critical,
                                      label=above:$s\le3$] {};
  \node[transition] (leave critical) [right=of critical] {};
  \node[transition] (enter critical) [left=of critical]  {};
\end{tikzpicture}
```

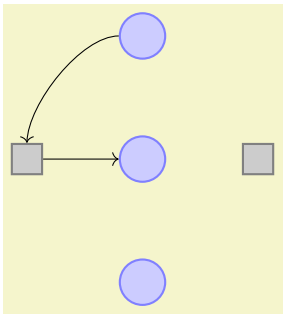
3.10 Connecting Nodes

It is now high time to connect the nodes. Let us start with something simple, namely with the straight line from `enter critical` to `critical`. We want this line to start at the right side of `enter critical` and to end at the left side of `critical`. For this, we can use the *anchors* of the nodes. Every node defines a whole bunch of anchors that lie on its border or inside it. For example, the `center` anchor is at the center of the node, the `west` anchor is on the left of the node, and so on. To access the coordinate of a node, we use a coordinate that contains the node's name followed by a dot, followed by the anchor's name:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
  \draw [->] (enter critical.east) -- (critical.west);
\end{tikzpicture}
```

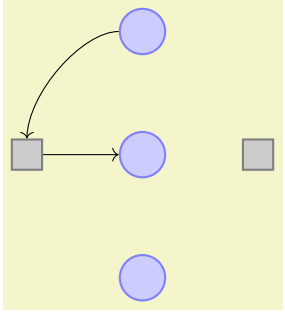
Next, let us tackle the curve from `waiting` to `enter critical`. This can be specified using curves and controls:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
  \draw [->] (enter critical.east) -- (critical.west);
  \draw [->] (waiting.west) .. controls +(left:5mm) and +(up:5mm)
                            .. (enter critical.north);
\end{tikzpicture}
```

Hagen sees how he can now add all his edges, but the whole process seems a but awkward and not very flexible. Again, the code seems to obscure the structure of the graphic rather than showing it.

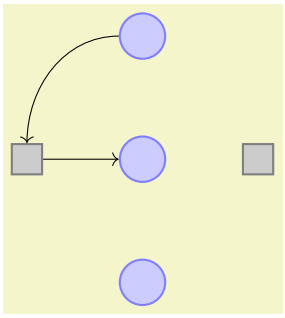
So, let us start improving the code for the edges. First, Hagen can leave out the anchors:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
  \draw [->] (enter critical) -- (critical);
  \draw [->] (waiting) .. controls +(left:8mm) and +(up:8mm)
                       .. (enter critical);
\end{tikzpicture}
```

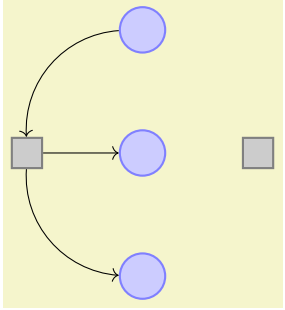
Hagen is a bit surprised that this works. After all, how did Ti*k*Z know that the line from `enter critical` to `critical` should actually start on the borders? Whenever Ti*k*Z encounters a whole node name as a "coordinate", it tries to "be smart" about the anchor that it should choose for this node. Depending on what happens next, Ti*k*Z will choose an anchor that lies on the border of the node on a line to the next coordinate or control point. The exact rules are a bit complex, but the chosen point will usually be correct – and when it is not, Hagen can still specify the desired anchor by hand.

Hagen would now like to simplify the curve operation somehow. It turns out that this can be accomplished using a special path operation: the `to` operation. This operation takes many options (you can even define new ones yourself). One pair of options is useful for Hagen: The pair `in` and `out`. These options take angles at which a curve should leave or reach the start or target coordinates. Without these options, a straight line is drawn:



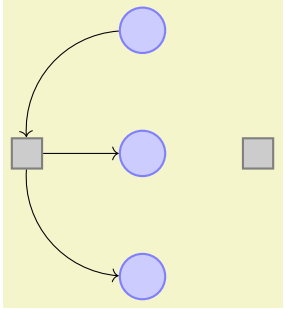
```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
  \draw [->] (enter critical) to                  (critical);
  \draw [->] (waiting)        to [out=180,in=90] (enter critical);
\end{tikzpicture}
```

There is another option for the `to` operation, that is even better suited to Hagen's problem: The `bend right` option. This option also takes an angle, but this angle only specifies the angle by which the curve is bent to the right:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {};
  \draw [->] (enter critical) to                  (critical);
  \draw [->] (waiting)        to [bend right=45] (enter critical);
  \draw [->] (enter critical) to [bend right=45] (semaphore);
\end{tikzpicture}
```

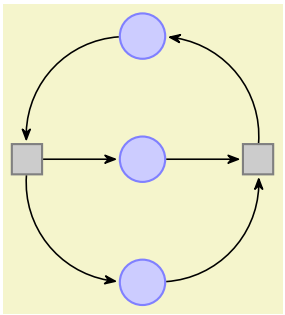
It is now time for Hagen to learn about yet another way of specifying edges: Using the `edge` path operation. This operation is very similar to the `to` operation, but there is one important difference: Like a node the edge generated by the `edge` operation is not part of the main path, but is added only later. This may not seem very important, but it has some nice consequences. For example, every edge can have its own arrow tips and its own color and so on and, still, all the edges can be given on the same path. This allows Hagen to write the following:



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};
  \node[transition] (leave critical) [right=of critical]   {};
  \node[transition] (enter critical) [left=of critical]    {}
    edge [->]                (critical)
    edge [<-,bend left=45]   (waiting)
    edge [->,bend right=45] (semaphore);
\end{tikzpicture}
```

Each `edge` caused a new path to be constructed, consisting of a `to` between the node `enter critical` and the node following the `edge` command.

The finishing touch is to introduce two styles `pre` and `post` and to use the `bend angle=45` option to set the bend angle once and for all:



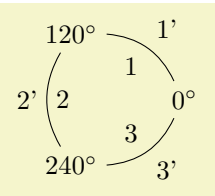
```
\usetikzlibrary {arrows.meta,positioning}
% Styles place and transition as before
\begin{tikzpicture}
  [bend angle=45,
   pre/.style={<-,shorten <=1pt,>={Stealth[round]},semithick},
   post/.style={->,shorten >=1pt,>={Stealth[round]},semithick}]

  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};

  \node[transition] (leave critical) [right=of critical]   {}
    edge [pre]              (critical)
    edge [post,bend right] (waiting)
    edge [pre, bend left]  (semaphore);
  \node[transition] (enter critical) [left=of critical]    {}
    edge [post]             (critical)
    edge [pre, bend left]  (waiting)
    edge [post,bend right] (semaphore);
\end{tikzpicture}
```

3.11 Adding Labels Next to Lines

The next thing that Hagen needs to add is the "2" at the arcs. For this Hagen can use Ti*k*Z's automatic node placement: By adding the option `auto`, Ti*k*Z will position nodes on curves and lines in such a way that they are not on the curve but next to it. Adding `swap` will mirror the label with respect to the line. Here is a general example:

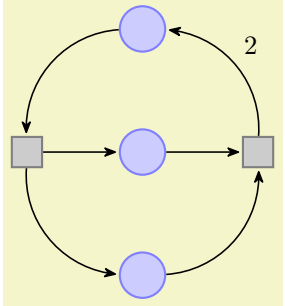



```
\begin{tikzpicture}[auto,bend right]
  \node (a) at (0:1) {$0^\circ$};
  \node (b) at (120:1) {$120^\circ$};
  \node (c) at (240:1) {$240^\circ$};

  \draw (a) to node {1} node [swap] {1'} (b)
        (b) to node {2} node [swap] {2'} (c)
        (c) to node {3} node [swap] {3'} (a);
\end{tikzpicture}
```

What is happening here? The nodes are given somehow inside the `to` operation! When this is done, the node is placed on the middle of the curve or line created by the `to` operation. The `auto` option then causes the node to be moved in such a way that it does not lie on the curve, but next to it. In the example we provide even two nodes on each `to` operation.

For Hagen that `auto` option is not really necessary since the two "2" labels could also easily be placed "by hand". However, in a complicated plot with numerous edges automatic placement can be a blessing.




```
\usetikzlibrary {arrows.meta,positioning}
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};

  \node[transition] (leave critical) [right=of critical]   {}
    edge [pre]                                  (critical)
    edge [post,bend right] node[auto,swap] {2} (waiting)
    edge [pre, bend left]                       (semaphore);
  \node[transition] (enter critical) [left=of critical]    {}
    edge [post]                                 (critical)
    edge [pre, bend left]                       (waiting)
    edge [post,bend right]                      (semaphore);
\end{tikzpicture}
```

3.12 Adding the Snaked Line and Multi-Line Text

With the node mechanism Hagen can now easily create the two Petri nets. What he is unsure of is how he can create the snaked line between the nets.

For this he can use a *decoration*. To draw the snaked line, Hagen only needs to set the two options `decoration=snake` and `decorate` on the path. This causes all lines of the path to be replaced by snakes. It is also possible to use snakes only in certain parts of a path, but Hagen will not need this.

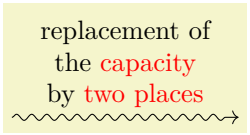


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,decoration=snake] (0,0) -- (2,0);
\end{tikzpicture}
```

Well, that does not look quite right, yet. The problem is that the snake happens to end exactly at the position where the arrow begins. Fortunately, there is an option that helps here. Also, the snake should be a bit smaller, which can be influenced by even more options.

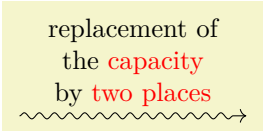

```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

Now Hagen needs to add the text above the snake. This text is a bit challenging since it is a multi-line text. Hagen has two options for this: First, he can specify an `align=center` and then use the `\\` command to enforce the line breaks at the desired positions.




```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0)
    node [above,align=center,midway]
    {
      replacement of\\
      the \textcolor{red}{capacity}\\
      by \textcolor{red}{two places}
    };
\end{tikzpicture}
```

Instead of specifying the line breaks "by hand", Hagen can also specify a width for the text and let TEX perform the line breaking for him:



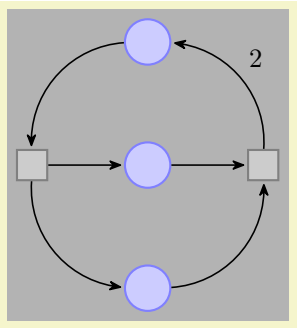

```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [->,decorate,
     decoration={snake,amplitude=.4mm,segment length=2mm,post length=1mm}]
    (0,0) -- (3,0)
    node [above,text width=3cm,align=center,midway]
    {
      replacement of the \textcolor{red}{capacity} by
      \textcolor{red}{two places}
    };
\end{tikzpicture}
```

3.13 Using Layers: The Background Rectangles

Hagen still needs to add the background rectangles. These are a bit tricky: Hagen would like to draw the rectangles *after* the Petri nets are finished. The reason is that only then can he conveniently refer to the coordinates that make up the corners of the rectangle. If Hagen draws the rectangle first, then he needs to know the exact size of the Petri net – which he does not.

The solution is to use *layers*. When the `backgrounds` library is loaded, Hagen can put parts of his picture inside a scope with the `on background layer` option. Then this part of the picture becomes part of the layer that is given as an argument to this environment. When the `{tikzpicture}` environment ends, the layers are put on top of each other, starting with the background layer. This causes everything drawn on the background layer to be behind the main text.

The next tricky question is, how big should the rectangle be? Naturally, Hagen can compute the size "by hand" or using some clever observations concerning the x- and y-coordinates of the nodes, but it would be nicer to just have Ti*k*Z compute a rectangle into which all the nodes "fit". For this, the `fit` library can be used. It defines the `fit` options, which, when given to a node, causes the node to be resized and shifted such that it exactly covers all the nodes and coordinates given as parameters to the `fit` option.




```
\usetikzlibrary {arrows.meta,backgrounds,fit,positioning}
% Styles as before
\begin{tikzpicture}[bend angle=45]
  \node[place]      (waiting)                              {};
  \node[place]      (critical)       [below=of waiting]    {};
  \node[place]      (semaphore)      [below=of critical]   {};

  \node[transition] (leave critical) [right=of critical]   {}
    edge [pre]                                  (critical)
    edge [post,bend right] node[auto,swap] {2}  (waiting)
    edge [pre, bend left]                       (semaphore);
  \node[transition] (enter critical) [left=of critical]    {}
    edge [post]                                 (critical)
    edge [pre, bend left]                       (waiting)
    edge [post,bend right]                      (semaphore);

  \begin{scope}[on background layer]
    \node [fill=black!30,fit=(waiting) (critical) (semaphore)
             (leave critical) (enter critical)] {};
  \end{scope}
\end{tikzpicture}
```

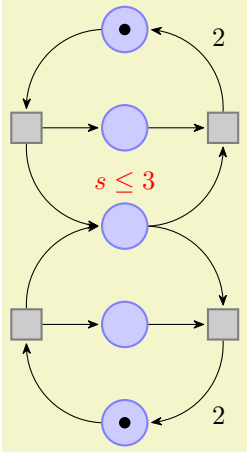
3.14 The Complete Code

Hagen has now finally put everything together. Only then does he learn that there is already a library for drawing Petri nets! It turns out that this library mainly provides the same definitions as Hagen did. For example, it defines a `place` style in a similar way as Hagen did. Adjusting the code so that it uses the library shortens Hagen code a bit, as shown in the following.

First, Hagen needs less style definitions, but he still needs to specify the colors of places and transitions.

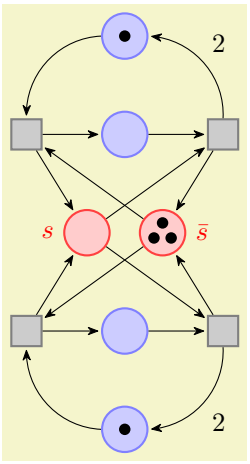
```
\begin{tikzpicture}
  [node distance=1.3cm,on grid,>={Stealth[round]},bend angle=45,auto,
   every place/.style=      {minimum size=6mm,thick,draw=blue!75,fill=blue!20},
   every transition/.style={thick,draw=black!75,fill=black!20},
   red place/.style=        {place,draw=red!75,fill=red!20},
   every label/.style=      {red}]
```

Now comes the code for the nets:




```
\usetikzlibrary {arrows.meta,petri,positioning}
   \node [place,tokens=1] (w1)                                  {};
   \node [place]          (c1) [below=of w1]                    {};
   \node [place]          (s)  [below=of c1,label=above:$s\le 3$] {};
   \node [place]          (c2) [below=of s]                     {};
   \node [place,tokens=1] (w2) [below=of c2]                    {};

   \node [transition] (e1) [left=of c1] {}
     edge [pre,bend left]                  (w1)
     edge [post,bend right]                (s)
     edge [post]                           (c1);
   \node [transition] (e2) [left=of c2] {}
     edge [pre,bend right]                 (w2)
     edge [post,bend left]                 (s)
     edge [post]                           (c2);
   \node [transition] (l1) [right=of c1] {}
     edge [pre]                            (c1)
     edge [pre,bend left]                  (s)
     edge [post,bend right] node[swap] {2} (w1);
   \node [transition] (l2) [right=of c2] {}
     edge [pre]                            (c2)
     edge [pre,bend right]                 (s)
     edge [post,bend left]  node {2}       (w2);
```




```
\usetikzlibrary {arrows.meta,petri,positioning}
  \begin{scope}[xshift=6cm]
    \node [place,tokens=1]      (w1')                                  {};
    \node [place]               (c1') [below=of w1']                   {};
    \node [red place]           (s1') [below=of c1',xshift=-5mm]
            [label=left:$s$]                                           {};
    \node [red place,tokens=3] (s2') [below=of c1',xshift=5mm]
            [label=right:$\bar s$]                                     {};
    \node [place]               (c2') [below=of s1',xshift=5mm]  {};
    \node [place,tokens=1]      (w2') [below=of c2']                   {};

    \node [transition] (e1') [left=of c1'] {}
      edge [pre,bend left]                  (w1')
      edge [post]                           (s1')
      edge [pre]                            (s2')
      edge [post]                           (c1');
    \node [transition] (e2') [left=of c2'] {}
      edge [pre,bend right]                 (w2')
      edge [post]                           (s1')
      edge [pre]                            (s2')
      edge [post]                           (c2');
    \node [transition] (l1') [right=of c1'] {}
      edge [pre]                            (c1')
      edge [pre]                            (s1')
      edge [post]                           (s2')
      edge [post,bend right] node[swap] {2} (w1');
    \node [transition] (l2') [right=of c2'] {}
      edge [pre]                            (c2')
      edge [pre]                            (s1')
      edge [post]                           (s2')
      edge [post,bend left]  node {2}       (w2');
  \end{scope}
```

The code for the background and the snake is the following:

```
  \begin{scope}[on background layer]
    \node (r1) [fill=black!10,rounded corners,fit=(w1)(w2)(e1)(e2)(l1)(l2)] {};
    \node (r2) [fill=black!10,rounded corners,fit=(w1')(w2')(e1')(e2')(l1')(l2')] {};
  \end{scope}

  \draw [shorten >=1mm,->,thick,decorate,
         decoration={snake,amplitude=.4mm,segment length=2mm,
                     pre=moveto,pre length=1mm,post length=2mm}]
    (r1) -- (r2) node [above=1mm,midway,text width=3cm,align=center]
      {replacement of the \textcolor{red}{capacity} by \textcolor{red}{two places}};
\end{tikzpicture}
```

4 Tutorial: Euclid's Amber Version of the *Elements*

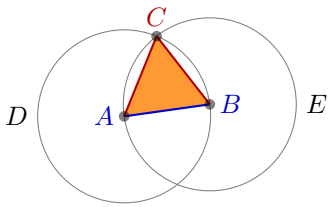
In this third tutorial we have a look at how Ti*k*Z can be used to draw geometric constructions.

Euclid is currently quite busy writing his new book series, whose working title is "Elements" (Euclid is not quite sure whether this title will convey the message of the series to future generations correctly, but he intends to change the title before it goes to the publisher). Up to now, he wrote down his text and graphics on papyrus, but his publisher suddenly insists that he must submit in electronic form. Euclid tries to argue with the publisher that electronics will only be discovered thousands of years later, but the publisher informs him that the use of papyrus is no longer cutting edge technology and Euclid will just have to keep up with modern tools.

Slightly disgruntled, Euclid starts converting his papyrus entitled "Book I, Proposition I" to an amber version.

4.1 Book I, Proposition I

The drawing on his papyrus looks like this:[1]



Proposition I
To construct an equilateral triangle on a given finite straight line.

Let AB be the given finite straight line. It is required to construct an equilateral triangle on the straight line AB.
Describe the circle BCD with center A and radius AB. Again describe the circle ACE with center B and radius BA. Join the straight lines CA and CB from the point C at which the circles cut one another to the points A and B.
Now, since the point A is the center of the circle CDB, therefore AC equals AB. Again, since the point B is the center of the circle CAE, therefore BC equals BA. But AC was proved equal to AB, therefore each of the straight lines AC and BC equals AB. And things which equal the same thing also equal one another, therefore AC also equals BC. Therefore the three straight lines AC, AB, and BC equal one another. Therefore the triangle ABC is equilateral, and it has been constructed on the given finite straight line AB.

Let us have a look at how Euclid can turn this into Ti*k*Z code.

4.1.1 Setting up the Environment

As in the previous tutorials, Euclid needs to load Ti*k*Z, together with some libraries. These libraries are `calc`, `intersections`, `through`, and `backgrounds`. Depending on which format he uses, Euclid would use one of the following in the preamble:

```
% For LaTeX:
\usepackage{tikz}
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For plain TeX:
\input{tikz.tex}
\usetikzlibrary{calc,intersections,through,backgrounds}
```

```
% For ConTeXt:
\usemodule[tikz]
\usetikzlibrary[calc,intersections,through,backgrounds]
```

[1] The text is taken from the wonderful interactive version of Euclid's Elements by David E. Joyce, to be found on his website at Clark University.

4.1.2 The Line AB

The first part of the picture that Euclid wishes to draw is the line AB. That is easy enough, something like `\draw (0,0) -- (2,1);` might do. However, Euclid does not wish to reference the two points A and B as $(0,0)$ and $(2,1)$ subsequently. Rather, he wishes to just write `A` and `B`. Indeed, the whole point of his book is that the points A and B can be arbitrary and all other points (like C) are constructed in terms of their positions. It would not do if Euclid were to write down the coordinates of C explicitly.

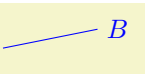
So, Euclid starts with defining two coordinates using the `\coordinate` command:



```
\begin{tikzpicture}
  \coordinate (A) at (0,0);
  \coordinate (B) at (1.25,0.25);

  \draw[blue] (A) -- (B);
\end{tikzpicture}
```

That was easy enough. What is missing at this point are the labels for the coordinates. Euclid does not want them *on* the points, but next to them. He decides to use the `label` option:




```
\begin{tikzpicture}
  \coordinate [label=left:\textcolor{blue}{$A$}]  (A) at (0,0);
  \coordinate [label=right:\textcolor{blue}{$B$}] (B) at (1.25,0.25);

  \draw[blue] (A) -- (B);
\end{tikzpicture}
```

At this point, Euclid decides that it would be even nicer if the points A and B were in some sense "random". Then, neither Euclid nor the reader can make the mistake of taking "anything for granted" concerning these position of these points. Euclid is pleased to learn that there is a `rand` function in Ti*k*Z that does exactly what he needs: It produces a number between -1 and 1. Since Ti*k*Z can do a bit of math, Euclid can change the coordinates of the points as follows:

```
\coordinate [...] (A) at (0+0.1*rand,0+0.1*rand);
\coordinate [...] (B) at (1.25+0.1*rand,0.25+0.1*rand);
```

This works fine. However, Euclid is not quite satisfied since he would prefer that the "main coordinates" $(0,0)$ and $(1.25,0.25)$ are "kept separate" from the perturbation $0.1(rand, rand)$. This means, he would like to specify that coordinate A as "the point that is at $(0,0)$ plus one tenth of the vector $(rand, rand)$".

It turns out that the `calc` library allows him to do exactly this kind of computation. When this library is loaded, you can use special coordinates that start with `($` and end with `$)` rather than just `(` and `)`. Inside these special coordinates you can give a linear combination of coordinates. (Note that the dollar signs are only intended to signal that a "computation" is going on; no mathematical typesetting is done.)

The new code for the coordinates is the following:

```
\coordinate [...] (A) at ($ (0,0) + .1*(rand,rand) $);
\coordinate [...] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);
```

Note that if a coordinate in such a computation has a factor (like `.1`), you must place a `*` directly before the opening parenthesis of the coordinate. You can nest such computations.

4.1.3 The Circle Around A

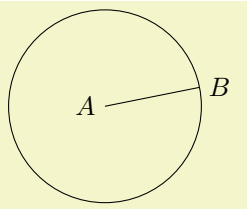
The first tricky construction is the circle around A. We will see later how to do this in a very simple manner, but first let us do it the "hard" way.

The idea is the following: We draw a circle around the point A whose radius is given by the length of the line AB. The difficulty lies in computing the length of this line.

Two ideas "nearly" solve this problem: First, we can write `($ (A) - (B) $)` for the vector that is the difference between A and B. All we need is the length of this vector. Second, given two numbers x and y, one can write `veclen(`x`,`y`)` inside a mathematical expression. This gives the value $\sqrt{x^2+y^2}$, which is exactly the desired length.

The only remaining problem is to access the x- and y-coordinate of the vector AB. For this, we need a new concept: the *let operation*. A let operation can be given anywhere on a path where a normal path operation like a line-to or a move-to is expected. The effect of a let operation is to evaluate some coordinates and to assign the results to special macros. These macros make it easy to access the x- and y-coordinates of the coordinates.

Euclid would write the following:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw (A) let
              \p1 = ($ (B) - (A) $)
            in
              circle [radius={veclen(\x1,\y1)}];
\end{tikzpicture}
```

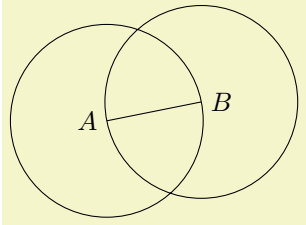
Each assignment in a let operation starts with `\p`, usually followed by a ⟨*digit*⟩. Then comes an equal sign and a coordinate. The coordinate is evaluated and the result is stored internally. From then on you can use the following expressions:

1. `\x`⟨*digit*⟩ yields the x-coordinate of the resulting point.
2. `\y`⟨*digit*⟩ yields the y-coordinate of the resulting point.
3. `\p`⟨*digit*⟩ yields the same as `\x`⟨*digit*⟩`,\y`⟨*digit*⟩.

You can have multiple assignments in a let operation, just separate them with commas. In later assignments you can already use the results of earlier assignments.

Note that `\p1` is not a coordinate in the usual sense. Rather, it just expands to a string like `10pt,20pt`. So, you cannot write, for instance, `(\p1.center)` since this would just expand to `(10pt,20pt.center)`, which makes no sense.

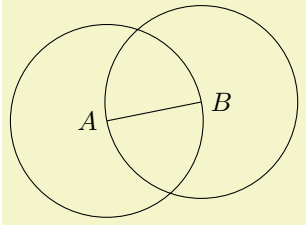
Next, we want to draw both circles at the same time. Each time the radius is `veclen(\x1,\y1)`. It seems natural to compute this radius only once. For this, we can also use a let operation: Instead of writing `\p1 = ...`, we write `\n2 = ...`. Here, "n" stands for "number" (while "p" stands for "point"). The assignment of a number should be followed by a number in curly braces.




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw let \p1 = ($ (B) - (A) $),
            \n2 = {veclen(\x1,\y1)}
        in
          (A) circle [radius=\n2]
          (B) circle [radius=\n2];
\end{tikzpicture}
```

In the above example, you may wonder, what `\n1` would yield? The answer is that it would be undefined – the `\p`, `\x`, and `\y` macros refer to the same logical point, while the `\n` macro has "its own namespace". We could even have replaced `\n2` in the example by `\n1` and it would still work. Indeed, the digits following these macros are just normal TEX parameters. We could also use a longer name, but then we have to use curly braces:



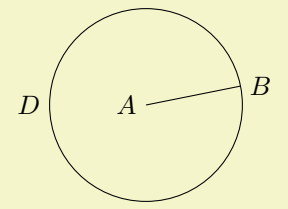

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \draw let \p1        = ($ (B) - (A) $),
            \n{radius} = {veclen(\x1,\y1)}
        in
          (A) circle [radius=\n{radius}]
          (B) circle [radius=\n{radius}];
\end{tikzpicture}
```

At the beginning of this section it was promised that there is an easier way to create the desired circle. The trick is to use the `through` library. As the name suggests, it contains code for creating shapes that go through a given point.

The option that we are looking for is `circle through`. This option is given to a *node* and has the following effects: First, it causes the node's inner and outer separations to be set to zero. Then it sets the

shape of the node to `circle`. Finally, it sets the radius of the node such that it goes through the parameter given to `circle through`. This radius is computed in essentially the same way as above.



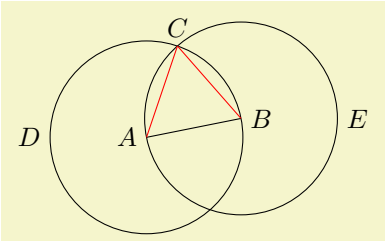

```
\usetikzlibrary {through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

  \node [draw,circle through=(B),label=left:$D$] at (A) {};
\end{tikzpicture}
```

4.1.4 The Intersection of the Circles

Euclid can now draw the line and the circles. The final problem is to compute the intersection of the two circles. This computation is a bit involved if you want to do it "by hand". Fortunately, the `intersections` library allows us to compute the intersection of arbitrary paths.

The idea is simple: First, you "name" two paths using the `name path` option. Then, at some later point, you can use the option `name intersections`, which creates coordinates called `intersection-1`, `intersection-2`, and so on at all intersections of the paths. Euclid assigns the names `D` and `E` to the paths of the two circles (which happen to be the same names as the nodes themselves, but nodes and their paths live in different "namespaces").




```
\usetikzlibrary {intersections,through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);

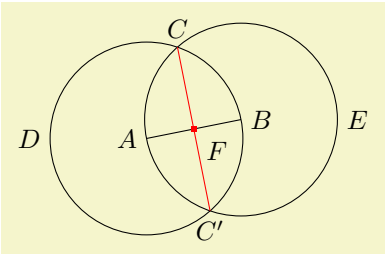
  \node (D) [name path=D,draw,circle through=(B),label=left:$D$]  at (A) {};
  \node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

  % Name the coordinates, but do not draw anything:
  \path [name intersections={of=D and E}];

  \coordinate [label=above:$C$] (C) at (intersection-1);

  \draw [red] (A) -- (C);
  \draw [red] (B) -- (C);
\end{tikzpicture}
```

It turns out that this can be further shortened: The `name intersections` takes an optional argument `by`, which lets you specify names for the coordinates and options for them. This creates more compact code. Although Euclid does not need it for the current picture, it is just a small step to computing the bisection of the line AB:



```
\usetikzlibrary {intersections,through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw [name path=A--B] (A) -- (B);

  \node (D) [name path=D,draw,circle through=(B),label=left:$D$]  at (A) {};
  \node (E) [name path=E,draw,circle through=(A),label=right:$E$] at (B) {};

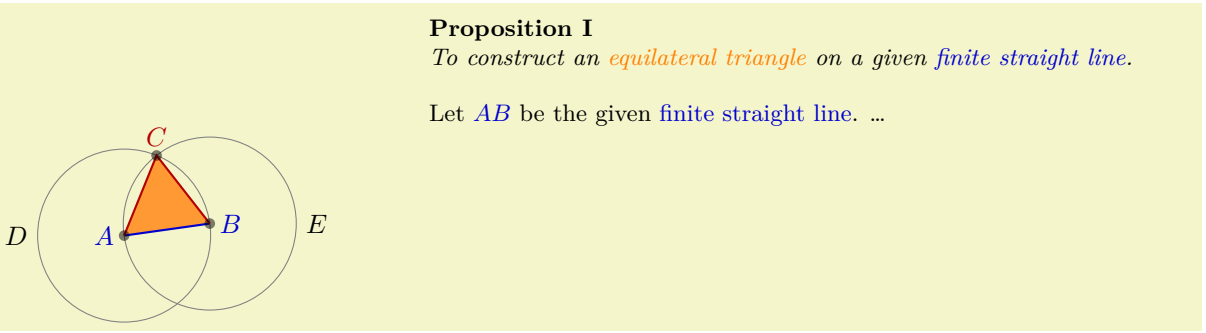
  \path [name intersections={of=D and E, by={[label=above:$C$]C, [label=below:$C'$]C'}}];

  \draw [name path=C--C',red] (C) -- (C');

  \path [name intersections={of=A--B and C--C',by=F}];
  \node [fill=red,inner sep=1pt,label=-45:$F$] at (F) {};
\end{tikzpicture}
```

4.1.5 The Complete Code

Back to Euclid's code. He introduces a few macros to make life simpler, like a `\A` macro for typesetting a blue A. He also uses the `background` layer for drawing the triangle behind everything at the end.



Proposition I
To construct an equilateral triangle on a given finite straight line.

Let AB be the given finite straight line. …

```
\usetikzlibrary {backgrounds,calc,intersections,through}
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
  \def\A{\textcolor{input}{$A$}}     \def\B{\textcolor{input}{$B$}}
  \def\C{\textcolor{output}{$C$}}    \def\D{$D$}
  \def\E{$E$}

  \colorlet{input}{blue!80!black}    \colorlet{output}{red!70!black}
  \colorlet{triangle}{orange}

  \coordinate [label=left:\A]  (A) at ($ (0,0) + .1*(rand,rand) $);
  \coordinate [label=right:\B] (B) at ($ (1.25,0.25) + .1*(rand,rand) $);

  \draw [input] (A) -- (B);

  \node [name path=D,help lines,draw,label=left:\D]   (D) at (A) [circle through=(B)] {};
  \node [name path=E,help lines,draw,label=right:\E]  (E) at (B) [circle through=(A)] {};

  \path [name intersections={of=D and E,by={[label=above:\C]C}}];

  \draw [output] (A) -- (C) -- (B);

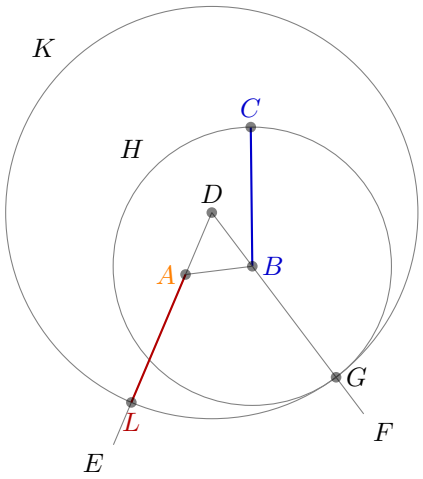
  \foreach \point in {A,B,C}
    \fill [black,opacity=.5] (\point) circle [radius=2pt];

  \begin{pgfonlayer}{background}
    \fill[triangle!80] (A) -- (C) -- (B) -- cycle;
  \end{pgfonlayer}

  \node [below right, text width=10cm,align=justify] at (4,3) {
    \small\textbf{Proposition I}\par
    \emph{To construct an \textcolor{triangle}{equilateral triangle}
      on a given \textcolor{input}{finite straight line}.}
    \par\vskip1em
    Let \A\B\ be the given \textcolor{input}{finite straight line}.  \dots
  };
\end{tikzpicture}
```

4.2 Book I, Proposition II

The second proposition in the Elements is the following:



Proposition II
To place a straight line equal to a given straight line with one end at a given point.

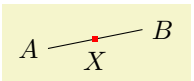
Let A be the given point, and BC the given straight line. It is required to place a straight line equal to the given straight line BC with one end at the point A.
Join the straight line AB from the point A to the point B, and construct the equilateral triangle DAB on it.
Produce the straight lines AE and BF in a straight line with DA and DB. Describe the circle CGH with center B and radius BC, and again, describe the circle GKL with center D and radius DG.
Since the point B is the center of the circle CGH, therefore BC equals BG. Again, since the point D is the center of the circle GKL, therefore DL equals DG. And in these DA equals DB, therefore the remainder AL equals the remainder BG. But BC was also proved equal to BG, therefore each of the straight lines AL and BC equals BG. And things which equal the same thing also equal one another, therefore AL also equals BC.
Therefore the straight line AL equal to the given straight line BC has been placed with one end at the given point A.

4.2.1 Using Partway Calculations for the Construction of *D*

Euclid's construction starts with "referencing" Proposition I for the construction of the point D. Now, while we could simply repeat the construction, it seems a bit bothersome that one has to draw all these circles and do all these complicated constructions.

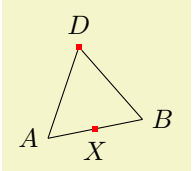
For this reason, TikZ supports some simplifications. First, there is a simple syntax for computing a point that is "partway" on a line from p to q: You place these two points in a coordinate calculation – remember, they start with `($` and end with `$)` – and then combine them using `!`⟨*part*⟩`!`. A ⟨*part*⟩ of `0` refers to the *first* coordinate, a ⟨*part*⟩ of `1` refers to the second coordinate, and a value in between refers to a point on the line from p to q. Thus, the syntax is similar to the `xcolor` syntax for mixing colors.

Here is the computation of the point in the middle of the line AB:



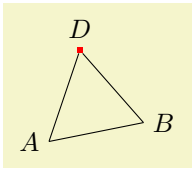

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)!.5!(B) $) {};
\end{tikzpicture}
```

The computation of the point D in Euclid's second proposition is a bit more complicated. It can be expressed as follows: Consider the line from X to B. Suppose we rotate this line around X for 90° and then stretch it by a factor of $\sin(60°) \cdot 2$. This yields the desired point D. We can do the stretching using the partway modifier above, for the rotation we need a new modifier: the rotation modifier. The idea is that the second coordinate in a partway computation can be prefixed by an angle. Then the partway point is computed normally (as if no angle were given), but the resulting point is rotated by this angle around the first point.




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=below:$X$] (X) at ($ (A)!.5!(B) $) {};
  \node [fill=red,inner sep=1pt,label=above:$D$] (D) at
    ($ (X) ! {sin(60)*2} ! 90:(B) $) {};
  \draw (A) -- (D) -- (B);
\end{tikzpicture}
```

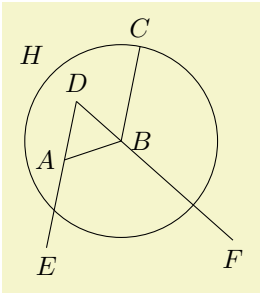
Finally, it is not necessary to explicitly name the point X. Rather, again like in the `xcolor` package, it is possible to chain partway modifiers:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (1.25,0.25);
  \draw (A) -- (B);
  \node [fill=red,inner sep=1pt,label=above:$D$] (D) at
    ($ (A) ! .5 ! (B) ! {sin(60)*2} ! 90:(B) $) {};
  \draw (A) -- (D) -- (B);
\end{tikzpicture}
```

4.2.2 Intersecting a Line and a Circle

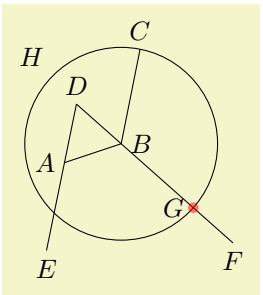
The next step in the construction is to draw a circle around B through C, which is easy enough to do using the `circle through` option. Extending the lines DA and DB can be done using partway calculations, but this time with a part value outside the range $[0, 1]$:




```
\usetikzlibrary {calc,through}
\begin{tikzpicture}
  \coordinate [label=left:$A$]  (A) at (0,0);
  \coordinate [label=right:$B$] (B) at (0.75,0.25);
  \coordinate [label=above:$C$] (C) at (1,1.5);
  \draw (A) -- (B) -- (C);
  \coordinate [label=above:$D$] (D) at
    ($ (A) ! .5 ! (B) ! {sin(60)*2} ! 90:(B) $) {};
  \node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
  \draw (D) -- ($ (D) ! 3.5 ! (B) $) coordinate [label=below:$F$] (F);
  \draw (D) -- ($ (D) ! 2.5 ! (A) $) coordinate [label=below:$E$] (E);
\end{tikzpicture}
```

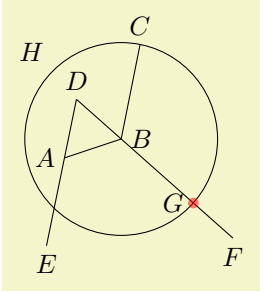
We now face the problem of finding the point G, which is the intersection of the line BF and the circle H. One way is to use yet another variant of the partway computation: Normally, a partway computation has the form $\langle p\rangle\,!\,\langle factor\rangle\,!\,\langle q\rangle$, resulting in the point $(1-\langle factor\rangle)\langle p\rangle+\langle factor\rangle\langle q\rangle$. Alternatively, instead of $\langle factor\rangle$ you can also use a $\langle dimension\rangle$ between the points. In this case, you get the point that is $\langle dimension\rangle$ away from $\langle p\rangle$ on the straight line to $\langle q\rangle$.

We know that the point G is on the way from B to F. The distance is given by the radius of the circle H. Here is the code for computing H:



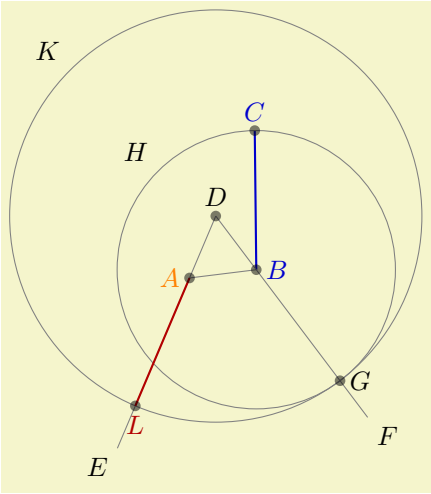

```
\usetikzlibrary {calc,through}
  \node (H) [label=135:$H$,draw,circle through=(C)] at (B) {};
  \path let \p1 = ($ (B) - (C) $) in
    coordinate [label=left:$G$] (G) at ($ (B) ! veclen(\x1,\y1) ! (F) $);
  \fill[red,opacity=.5] (G) circle [radius=2pt];
```

However, there is a simpler way: We can simply name the path of the circle and of the line in question and then use `name intersections` to compute the intersections.




```
\usetikzlibrary {calc,intersections,through}
  \node (H) [name path=H,label=135:$H$,draw,circle through=(C)] at (B) {};
  \path [name path=B--F] (B) -- (F);
  \path [name intersections={of=H and B--F,by={[label=left:$G$]G}}];
  \fill[red,opacity=.5] (G) circle [radius=2pt];
```

4.2.3 The Complete Code




```
\usetikzlibrary {calc,intersections,through}
\begin{tikzpicture}[thick,help lines/.style={thin,draw=black!50}]
  \def\A{\textcolor{orange}{$A$}}   \def\B{\textcolor{input}{$B$}}
  \def\C{\textcolor{input}{$C$}}    \def\D{$D$}
  \def\E{$E$}                       \def\F{$F$}
  \def\G{$G$}                       \def\H{$H$}
  \def\K{$K$}                       \def\L{\textcolor{output}{$L$}}

  \colorlet{input}{blue!80!black}    \colorlet{output}{red!70!black}

  \coordinate [label=left:\A]  (A) at ($ (0,0) + .1*(rand,rand) $);
  \coordinate [label=right:\B] (B) at ($ (1,0.2) + .1*(rand,rand) $);
  \coordinate [label=above:\C] (C) at ($ (1,2) + .1*(rand,rand) $);

  \draw [input] (B) -- (C);
  \draw [help lines] (A) -- (B);

  \coordinate [label=above:\D] (D) at ($ (A)!.5!(B) ! {sin(60)*2} ! 90:(B) $);

  \draw [help lines] (D) -- ($ (D)!3.75!(A) $) coordinate [label=-135:\E] (E);
  \draw [help lines] (D) -- ($ (D)!3.75!(B) $) coordinate [label=-45:\F] (F);

  \node (H) at (B) [name path=H,help lines,circle through=(C),draw,label=135:\H] {};
  \path [name path=B--F] (B) -- (F);
  \path [name intersections={of=H and B--F,by={[label=right:\G]G}}];

  \node (K) at (D) [name path=K,help lines,circle through=(G),draw,label=135:\K] {};
  \path [name path=A--E] (A) -- (E);
  \path [name intersections={of=K and A--E,by={[label=below:\L]L}}];

  \draw [output] (A) -- (L);

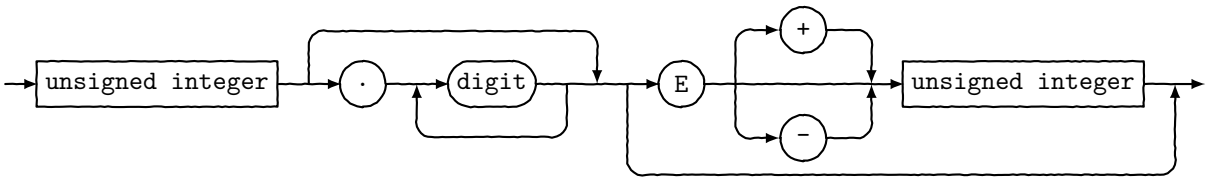
  \foreach \point in {A,B,C,D,G,L}
    \fill [black,opacity=.5] (\point) circle [radius=2pt];

  % \node ...
\end{tikzpicture}
```

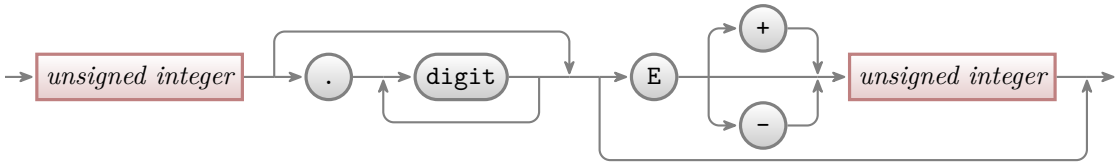
5 Tutorial: Diagrams as Simple Graphs

In this tutorial we have a look at how graphs and matrices can be used to typeset a diagram.

Ilka, who just got tenure for her professorship on Old and Lovable Programming Languages, has recently dug up a technical report entitled *The Programming Language Pascal* in the dusty cellar of the library of her university. Having been created in the good old times using pens and rules, it looks like this[2]:



For her next lecture, Ilka decides to redo this diagram, but this time perhaps a bit cleaner and perhaps also bit "cooler".



Having read the previous tutorials, Ilka knows already how to set up the environment for her diagram, namely using a `tikzpicture` environment. She wonders which libraries she will need. She decides that she will postpone the decision and add the necessary libraries as needed as she constructs the picture.

5.1 Styling the Nodes

The bulk of this tutorial will be about arranging the nodes and connecting them using chains, but let us start with setting up styles for the nodes.

There are two kinds of nodes in the diagram, namely what theoreticians like to call *terminals* and *nonterminals*. For the terminals, Ilka decides to use a black color, which visually shows that "nothing needs to be done about them". The nonterminals, which still need to be "processed" further, get a bit of red mixed in.

Ilka starts with the simpler nonterminals, as there are no rounded corners involved. Naturally, she sets up a style:

unsigned integer

```
\usetikzlibrary {positioning}
\begin{tikzpicture}[
    nonterminal/.style={
      % The shape:
      rectangle,
      % The size:
      minimum size=6mm,
      % The border:
      very thick,
      draw=red!50!black!50,          % 50% red and 50% black,
                                     % and that mixed with 50% white
      % The filling:
      top color=white,               % a shading that is white at the top...
      bottom color=red!50!black!20, % and something else at the bottom
      % Font
      font=\itshape
    }]
  \node [nonterminal] {unsigned integer};
\end{tikzpicture}
```

Ilka is pretty proud of the use of the `minimum size` option: As the name suggests, this option ensures that the node is at least 6mm by 6mm, but it will expand in size as necessary to accommodate longer text. By giving this option to all nodes, they will all have the same height of 6mm.

Styling the terminals is a bit more difficult because of the round corners. Ilka has several options how she can achieve them. One way is to use the `rounded corners` option. It gets a dimension as parameter

[2]The shown diagram was not scanned, but rather typeset using Ti*k*Z. The jittering lines were created using the `random steps` decoration.

and causes all corners to be replaced by little arcs with the given dimension as radius. By setting the radius to 3mm, she will get exactly what she needs: circles, when the shapes are, indeed, exactly 6mm by 6mm and otherwise half circles on the sides:




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[node distance=5mm,
                    terminal/.style={
                      % The shape:
                      rectangle,minimum size=6mm,rounded corners=3mm,
                      % The rest
                      very thick,draw=black!50,
                      top color=white,bottom color=black!20,
                      font=\ttfamily}]
  \node (dot)   [terminal]                {.};
  \node (digit) [terminal,right=of dot]   {digit};
  \node (E)     [terminal,right=of digit] {E};
\end{tikzpicture}
```

Another possibility is to use a shape that is specially made for typesetting rectangles with arcs on the sides (she has to use the `shapes.misc` library to use it). This shape gives Ilka much more control over the appearance. For instance, she could have an arc only on the left side, but she will not need this.




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm,
                    terminal/.style={
                      % The shape:
                      rounded rectangle,
                      minimum size=6mm,
                      % The rest
                      very thick,draw=black!50,
                      top color=white,bottom color=black!20,
                      font=\ttfamily}]
  \node (dot)   [terminal]                {.};
  \node (digit) [terminal,right=of dot]   {digit};
  \node (E)     [terminal,right=of digit] {E};
\end{tikzpicture}
```

At this point, she notices a problem. The baseline of the text in the nodes is not aligned:




```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm]
  \node (dot)   [terminal]                {.};
  \node (digit) [terminal,right=of dot]   {digit};
  \node (E)     [terminal,right=of digit] {E};

  \draw [help lines] let \p1 = (dot.base),
                         \p2 = (digit.base),
                         \p3 = (E.base)
                     in (-.5,\y1) -- (3.5,\y1)
                        (-.5,\y2) -- (3.5,\y2)
                        (-.5,\y3) -- (3.5,\y3);
\end{tikzpicture}
```

(Ilka has moved the style definition to the preamble by saying `\tikzset{terminal/.style=...}`, so that she can use it in all pictures.)

For the `digit` and the `E` the difference in the baselines is almost imperceptible, but for the dot the problem is quite severe: It looks more like a multiplication dot than a period.

Ilka toys with the idea of using the `base right=of...` option rather than `right=of...` to align the nodes in such a way that the baselines are all on the same line (the `base right` option places a node right of something so that the baseline is right of the baseline of the other object). However, this does not have the desired effect:




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm]
  \node (dot)   [terminal]                     {.};
  \node (digit) [terminal,base right=of dot]   {digit};
  \node (E)     [terminal,base right=of digit] {E};
\end{tikzpicture}
```

The nodes suddenly “dance around”! There is no hope of changing the position of text inside a node using anchors. Instead, Ilka must use a trick: The problem of mismatching baselines is caused by the fact

that `.` and `digit` and `E` all have different heights and depth. If they all had the same, they would all be positioned vertically in the same manner. So, all Ilka needs to do is to use the `text height` and `text depth` options to explicitly specify a height and depth for the nodes.




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm,
                    text height=1.5ex,text depth=.25ex]
  \node (dot)    [terminal]                {.};
  \node (digit)  [terminal,right=of dot]   {digit};
  \node (E)      [terminal,right=of digit] {E};
\end{tikzpicture}
```

5.2 Aligning the Nodes Using Positioning Options

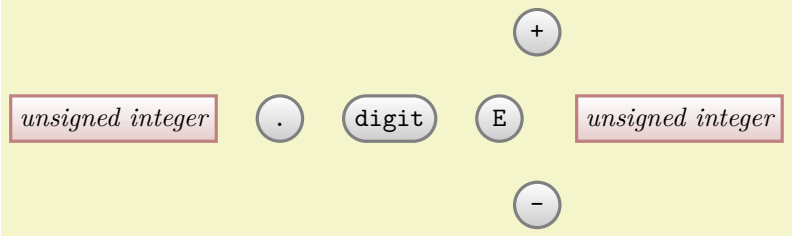
Ilka now has the "styling" of the nodes ready. The next problem is to place them in the right places. There are several ways to do this. The most straightforward is to simply explicitly place the nodes at certain coordinates "calculated by hand". For very simple graphics this is perfectly alright, but it has several disadvantages:

1. For more difficult graphics, the calculation may become complicated.
2. Changing the text of the nodes may make it necessary to recalculate the coordinates.
3. The source code of the graphic is not very clear since the relationships between the positions of the nodes are not made explicit.

For these reasons, Ilka decides to try out different ways of arranging the nodes on the page.

The first method is the use of *positioning options*. To use them, you need to load the `positioning` library. This gives you access to advanced implementations of options like `above` or `left`, since you can now say `above=of some node` in order to place a node above of `some node`, with the borders separated by `node distance`.

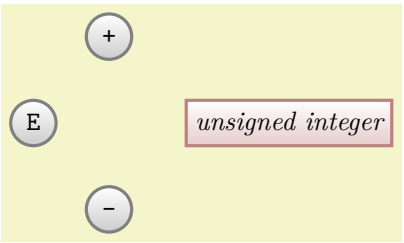
Ilka can use this to draw the place the nodes in a long row:




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm]
  \node (ui1)    [nonterminal]                    {unsigned integer};
  \node (dot)    [terminal,right=of ui1]          {.};
  \node (digit)  [terminal,right=of dot]          {digit};
  \node (E)      [terminal,right=of digit]        {E};
  \node (plus)   [terminal,above right=of E]      {+};
  \node (minus)  [terminal,below right=of E]      {-};
  \node (ui2)    [nonterminal,below right=of plus] {unsigned integer};
\end{tikzpicture}
```

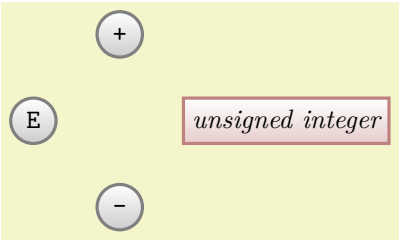
For the plus and minus nodes, Ilka is a bit startled by their placements. Shouldn't they be more to the right? The reason they are placed in that manner is the following: The `north east` anchor of the `E` node lies at the "upper start of the right arc", which, a bit unfortunately in this case, happens to be the top of the node. Likewise, the `south west` anchor of the `+` node is actually at its bottom and, indeed, the horizontal and vertical distances between the top of the `E` node and the bottom of the `+` node are both 5mm.

There are several ways of fixing this problem. The easiest way is to simply add a little bit of horizontal shift by hand:




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm]
  \node (E)      [terminal]                                  {E};
  \node (plus)   [terminal,above right=of E,xshift=5mm]      {+};
  \node (minus)  [terminal,below right=of E,xshift=5mm]      {-};
  \node (ui2)    [nonterminal,below right=of plus,xshift=5mm] {unsigned integer};
\end{tikzpicture}
```

A second way is to revert back to the idea of using a normal rectangle for the terminals, but with rounded corners. Since corner rounding does not affect anchors, she gets the following result:




```
\usetikzlibrary {positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,terminal/.append style={rectangle,rounded corners=3mm}]
  \node (E)      [terminal]                          {E};
  \node (plus)   [terminal,above right=of E]         {+};
  \node (minus)  [terminal,below right=of E]         {-};
  \node (ui2)    [nonterminal,below right=of plus]   {unsigned integer};
\end{tikzpicture}
```

A third way is to use matrices, which we will do later.

Now that the nodes have been placed, Ilka needs to add connections. Here, some connections are more difficult than others. Consider for instance the "repeat" line around the `digit`. One way of describing this line is to say "it starts a little to the right of `digit` than goes down and then goes to the left and finally ends at a point a little to the left of `digit`". Ilka can put this into code as follows:




```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm]
  \node (dot)    [terminal]                          {.};
  \node (digit)  [terminal,right=of dot]             {digit};
  \node (E)      [terminal,right=of digit]           {E};

  \path (dot)    edge[->] (digit)  % simple edges
        (digit)  edge[->] (E);

  \draw [->]
     % start right of digit.east, that is, at the point that is the
     % linear combination of digit.east and the vector (2mm,0pt). We
     % use the ($ ... $) notation for computing linear combinations
     ($ (digit.east) + (2mm,0) $)
     % Now go down
     -- ++(0,-.5)
     % And back to the left of digit.west
     -| ($ (digit.west) - (2mm,0) $);
\end{tikzpicture}
```

Since Ilka needs this "go up/down then horizontally and then up/down to a target" several times, it seems sensible to define a special *to-path* for this. Whenever the `edge` command is used, it simply adds the current value of `to path` to the path. So, Ilka can set up a style that contains the correct path:




```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,
    skip loop/.style={to path={-- ++(0,-.5) -| (\tikztotarget)}}]
  \node (dot)   [terminal]                        {.};
  \node (digit) [terminal,right=of dot]           {digit};
  \node (E)     [terminal,right=of digit]         {E};

  \path (dot)   edge[->]           (digit)  % simple edges
        (digit) edge[->]           (E)
        ($ (digit.east) + (2mm,0) $)
                edge[->,skip loop] ($ (digit.west) - (2mm,0) $);
\end{tikzpicture}
```

Ilka can even go a step further and make her `skip loop` style parameterized. For this, the skip loop's vertical offset is passed as parameter `#1`. Also, in the following code Ilka specifies the start and targets differently, namely as the positions that are "in the middle between the nodes".




```
\usetikzlibrary {calc,positioning,shapes.misc}
\begin{tikzpicture}[node distance=5mm and 5mm,
    skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}]
  \node (dot)   [terminal]                        {.};
  \node (digit) [terminal,right=of dot]           {digit};
  \node (E)     [terminal,right=of digit]         {E};

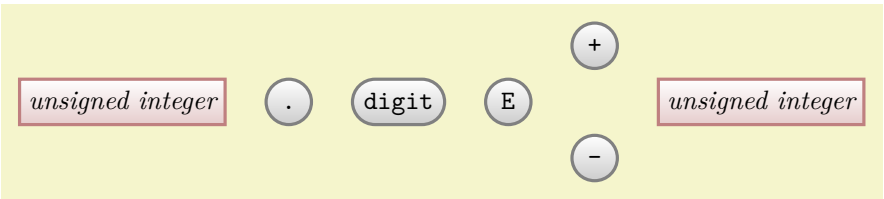
  \path (dot)   edge[->]               (digit)  % simple edges
        (digit) edge[->]               (E)
        ($ (digit.east)!.5!(E.west) $)
                edge[->,skip loop=-5mm] ($ (digit.west)!.5!(dot.east) $);
\end{tikzpicture}
```

5.3 Aligning the Nodes Using Matrices

Ilka is still bothered a bit by the placement of the plus and minus nodes. Somehow, having to add an explicit `xshift` seems too much like cheating.

A perhaps better way of positioning the nodes is to use a *matrix*. In Ti*k*Z matrices can be used to align quite arbitrary graphical objects in rows and columns. The syntax is very similar to the use of arrays and tables in TEX (indeed, internally TEX tables are used, but a lot of stuff is going on additionally).

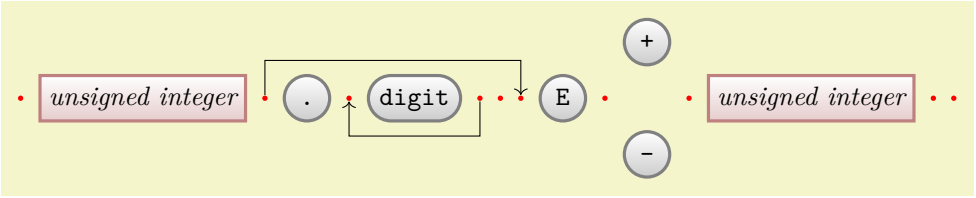
In Ilka's graphic, there will be three rows: One row containing only the plus node, one row containing the main nodes and one row containing only the minus node.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \matrix[row sep=1mm,column sep=5mm] {
    % First row:
      & & & & \node [terminal] {+}; & \\
    % Second row:
    \node [nonterminal] {unsigned integer}; &
    \node [terminal]    {.};                &
    \node [terminal]    {digit};            &
    \node [terminal]    {E};                &
                                            &
    \node [nonterminal] {unsigned integer}; \\
    % Third row:
      & & & & \node [terminal] {-}; & \\
  };
\end{tikzpicture}
```

That was easy! By toying around with the row and columns separations, Ilka can achieve all sorts of pleasing arrangements of the nodes.

Ilka now faces the same connecting problem as before. This time, she has an idea: She adds small nodes (they will be turned into coordinates later on and be invisible) at all the places where she would like connections to start and end.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}[point/.style={circle,inner sep=0pt,minimum size=2pt,fill=red},
                    skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}}]
  \matrix[row sep=1mm,column sep=2mm] {
    % First row:
    & & & & & & & & & & \node (plus) [terminal] {+};\\
    % Second row:
    \node (p1) [point]  {}; &    \node (ui1)   [nonterminal] {unsigned integer}; &
    \node (p2) [point]  {}; &    \node (dot)   [terminal]    {.};                &
    \node (p3) [point]  {}; &    \node (digit) [terminal]    {digit};            &
    \node (p4) [point]  {}; &    \node (p5)    [point]  {};                       &
    \node (p6) [point]  {}; &    \node (e)     [terminal]    {E};                &
    \node (p7) [point]  {}; &                                                     &
    \node (p8) [point]  {}; &    \node (ui2)   [nonterminal] {unsigned integer}; &
    \node (p9) [point]  {}; &    \node (p10)   [point]       {};\\
    % Third row:
    & & & & & & & & & & \node (minus)[terminal] {-};\\
  };

  \path (p4) edge [->,skip loop=-5mm] (p3)
        (p2) edge [->,skip loop=5mm]  (p6);
\end{tikzpicture}
```

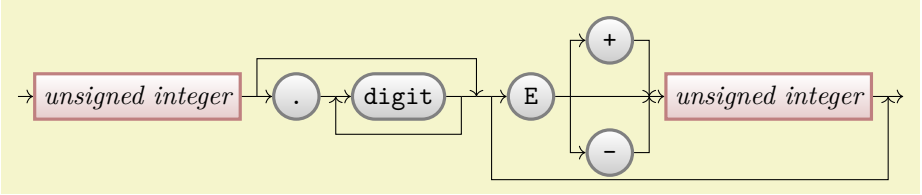
Now, it's only a small step to add all the missing edges.

5.4 The Diagram as a Graph

Matrices allow Ilka to align the nodes nicely, but the connections are not quite perfect. The problem is that the code does not really reflect the paths that underlie the diagram. For this, it seems natural enough to Ilka to use the `graphs` library since, after all, connecting nodes by edges is exactly what happens in a graph. The `graphs` library can both be used to connect nodes that have already been created, but it can also be used to create nodes "on the fly" and these processes can also be mixed.

5.4.1 Connecting Already Positioned Nodes

Ilka has already a fine method for positioning her nodes (using a `matrix`), so all that she needs is an easy way of specifying the edges. For this, she uses the `\graph` command (which is actually just a shorthand for `\path graph`). It allows her to write down edges between them in a simple way (the macro `\matrixcontent` contains exactly the matrix content from the previous example; no need to repeat it here):



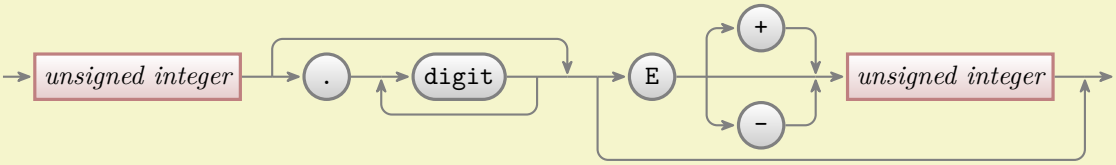
```
\usetikzlibrary {graphs,shapes.misc}
\begin{tikzpicture}[skip loop/.style={to path={-- ++(0,#1) -| (\tikztotarget)}},
                    hv path/.style={to path={-| (\tikztotarget)}},
                    vh path/.style={to path={|- (\tikztotarget)}}]
  \matrix[row sep=1mm,column sep=2mm] { \matrixcontent };

  \graph {
    (p1) -> (ui1) -- (p2) -> (dot) -- (p3) -> (digit) -- (p4)
         -- (p5)  -- (p6) -> (e) -- (p7) -- (p8) -> (ui2) -- (p9) -> (p10);
    (p4) ->[skip loop=-5mm]  (p3);
    (p2) ->[skip loop=5mm]   (p5);
    (p6) ->[skip loop=-11mm] (p9);
    (p7) ->[vh path]         (plus)  -> [hv path] (p8);
    (p7) ->[vh path]         (minus) -> [hv path] (p8);
  };
\end{tikzpicture}
```

This is already pretty near to the desired result, just a few "finishing touches" are needed to style the edges more nicely.

However, Ilka does not have the feeling that the `graph` command is all that hot in the example. It certainly does cut down on the number of characters she has to write, but the overall graph structure is not that much clear – it is still mainly a list of paths through the graph. It would be nice to specify that, say, there the path from `(p7)` sort of splits to `(plus)` and `(minus)` and then merges once more at `(p8)`. Also, all these parentheses are bit hard to type.

It turns out that edges from a node to a whole group of nodes are quite easy to specify, as shown in the next example. Additionally, by using the `use existing nodes` option, Ilka can also leave out all the parentheses (again, some options have been moved outside to keep the examples shorter):

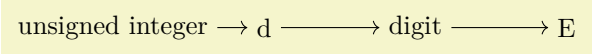



```
\usetikzlibrary {arrows.meta,graphs,shapes.misc}
\begin{tikzpicture}[>={Stealth[round]},thick,black!50,text=black,
                    every new ->/.style={shorten >=1pt},
                    graphs/every graph/.style={edges=rounded corners}]
  \matrix[column sep=4mm] { \matrixcontent };

  \graph [use existing nodes] {
    p1 -> ui1 -- p2 -> dot -- p3 -> digit -- p4 -- p5  -- p6 -> e -- p7 -- p8 -> ui2 -- p9 -> p10;
    p4 ->[skip loop=-5mm]  p3;
    p2 ->[skip loop=5mm]   p5;
    p6 ->[skip loop=-11mm] p9;
    p7 ->[vh path] { plus, minus } -> [hv path] p8;
  };
\end{tikzpicture}
```

5.4.2 Creating Nodes Using the Graph Command

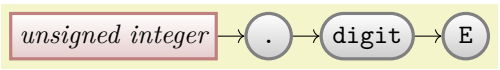
Ilka has heard that the `graph` command is also supposed to make it easy to create nodes, not only to connect them. This is, indeed, correct: When the `use existing nodes` option is not used and when a node name is not surrounded by parentheses, then Ti*k*Z will actually create a node whose name and text is the node name:




```
\usetikzlibrary {graphs}
\tikz \graph [grow right=2cm] { unsigned integer -> d -> digit -> E };
```

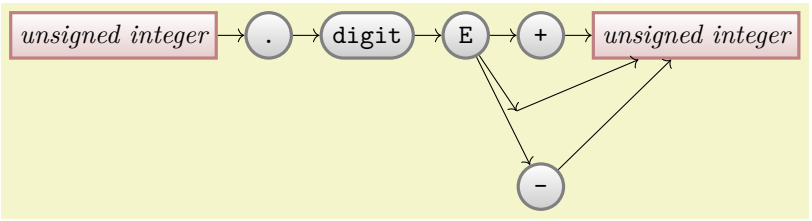
Not quite perfect, but we are getting somewhere. First, let us change the positioning algorithm by saying `grow right sep`, which causes new nodes to be placed to the right of the previous nodes with a certain fixed separation (`1em` by default). Second, we add some options to make the node "look nice". Third, note

the funny `d` node above: Ilka tried writing just `.` there first, but got some error messages. The reason is that a node cannot be called `.` in Ti*k*Z, so she had to choose a different name – which is not good, since she wants a dot to be shown! The trick is to put the dot in quotation marks, this allows you to use "quite arbitrary text" as a node name:




```
\usetikzlibrary {graphs,shapes.misc}
\tikz \graph [grow right sep] {
  unsigned integer[nonterminal] -> "."[terminal] -> digit[terminal] -> E[terminal]
};
```

Now comes the fork to the plus and minus signs. Here, Ilka can use the grouping mechanism of the `graph` command to create a split:

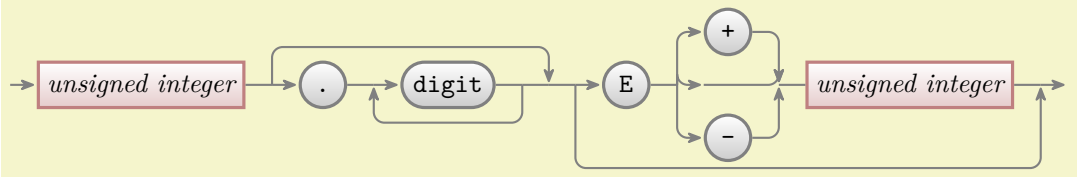



```
\usetikzlibrary {graphs,shapes.misc}
\tikz \graph [grow right sep] {
  unsigned integer  [nonterminal] ->
  "."               [terminal] ->
  digit             [terminal] ->
  E                 [terminal] ->
  {
    "+"             [terminal],
    ""              [coordinate],
    "-"             [terminal]
  } ->
  ui2/unsigned integer [nonterminal]
};
```

Let us see, what is happening here. We want two `unsigned integer` nodes, but if we just were to use this text twice, then Ti*k*Z would have noticed that the same name was used already in the current graph and, being smart (actually too smart in this case), would have created an edge back to the already-created node. Thus, a fresh name is needed here. However, Ilka also cannot just write `unsigned integer2`, because she wants the original text to be shown, after all! The trick is to use a slash inside the node name: In order to "render" the node, the text following the slash is used instead of the node name, which is the text before the slash. Alternatively, the `as` option can be used, which also allows you to specify how a node should be rendered.

It turns out that Ilka does not need to invent a name like `ui2` for a node that she will not reference again anyway. In this case, she can just leave out the name (write nothing before `/`), which always stands for a "fresh, anonymous" node name.

Next, Ilka needs to add some coordinates in between of some nodes where the back-loops should got and she needs to shift the nodes a bit:

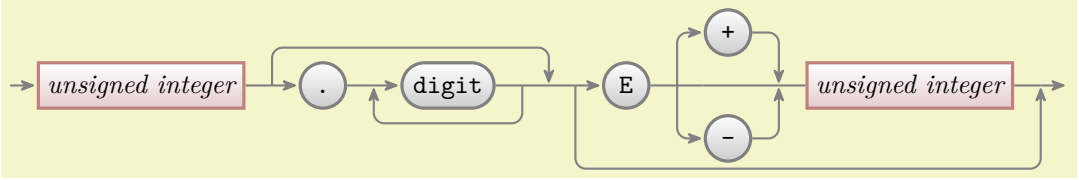


```
\usetikzlibrary {arrows.meta,graphs,shapes.misc}
\begin{tikzpicture}[>={Stealth[round]}, thick, black!50, text=black,
                    every new ->/.style={shorten >=1pt},
                    graphs/every graph/.style={edges=rounded corners}]
  \graph [grow right sep, branch down=7mm] {
    /                   [coordinate] ->
    unsigned integer    [nonterminal] --
    p1                  [coordinate] ->
    "."                 [terminal] --
    p2                  [coordinate] ->
    digit               [terminal] --
    p3                  [coordinate] --
    p4                  [coordinate] --
    p5                  [coordinate] ->
    E                   [terminal] --
    q1                  [coordinate] ->[vh path]
    { [nodes={yshift=7mm}]
      "+"                 [terminal],
      q2/                 [coordinate],
      "-"                 [terminal]
    } -> [hv path]
    q3                  [coordinate] --
    /unsigned integer   [nonterminal] --
    p6                  [coordinate] ->
    /                   [coordinate];

    p1 ->[skip loop=5mm]   p4;
    p3 ->[skip loop=-5mm]  p2;
    p5 ->[skip loop=-11mm] p6;
  };
\end{tikzpicture}
```

All that remains to be done is to somehow get rid of the strange curves between the `E` and the unsigned integer. They are caused by Ti*k*Z's attempt at creating an edge that first goes vertical and then horizontal but is actually just horizontal. Additionally, the edge should not really be pointed; but it seems difficult to get rid of this since the *other* edges from `q1`, namely to `plus` and `minus` should be pointed.

It turns out that there is a nice way of solving this problem: You can specify that a graph is `simple`. This means that there can be at most one edge between any two nodes. Now, if you specify an edge twice, the options of the second specification "win". Thus, by adding two more lines that "correct" these edges, we get the final diagram with its complete code:



```
\usetikzlibrary {arrows.meta,graphs,shapes.misc}
\tikz [>={Stealth[round]}, black!50, text=black, thick,
       every new ->/.style          = {shorten >=1pt},
       graphs/every graph/.style    = {edges=rounded corners},
       skip loop/.style             = {to path={-- ++(0,#1) -| (\tikztotarget)}},
       hv path/.style               = {to path={-| (\tikztotarget)}},
       vh path/.style               = {to path={|- (\tikztotarget)}},
       nonterminal/.style           = {
         rectangle, minimum size=6mm, very thick, draw=red!50!black!50, top color=white,
         bottom color=red!50!black!20, font=\itshape, text height=1.5ex,text depth=.25ex},
       terminal/.style              = {
         rounded rectangle,  minimum size=6mm, very thick, draw=black!50, top color=white,
         bottom color=black!20, font=\ttfamily, text height=1.5ex, text depth=.25ex},
       shape                        = coordinate
       ]
  \graph [grow right sep, branch down=7mm, simple] {
    / -> unsigned integer[nonterminal] -- p1 -> "." [terminal] -- p2 -> digit[terminal] --
    p3 -- p4 -- p5 -> E[terminal] -- q1 ->[vh path]
    {[nodes={yshift=7mm}]
      "+"[terminal], q2, "-"[terminal]
    } -> [hv path]
    q3 -- /unsigned integer [nonterminal] -- p6 -> /;

    p1 ->[skip loop=5mm]   p4;
    p3 ->[skip loop=-5mm]  p2;
    p5 ->[skip loop=-11mm] p6;

    q1 -- q2 -- q3;  % make these edges plain
  };
```

6 Tutorial: A Lecture Map for Johannes

In this tutorial we explore the tree and mind map mechanisms of Ti*k*Z.

Johannes is quite excited: For the first time he will be teaching a course all by himself during the upcoming semester! Unfortunately, the course is not on his favorite subject, which is of course Theoretical Immunology, but on Complexity Theory, but as a young academic Johannes is not likely to complain too loudly. In order to help the students get a general overview of what is going to happen during the course as a whole, he intends to draw some kind of tree or graph containing the basic concepts. He got this idea from his old professor who seems to be using these "lecture maps" with some success. Independently of the success of these maps, Johannes thinks they look quite neat.

6.1 Problem Statement

Johannes wishes to create a lecture map with the following features:

1. It should contain a tree or graph depicting the main concepts.
2. It should somehow visualize the different lectures that will be taught. Note that the lectures are not necessarily the same as the concepts since the graph may contain more concepts than will be addressed in lectures and some concepts may be addressed during more than one lecture.
3. The map should also contain a calendar showing when the individual lectures will be given.
4. The aesthetical reasons, the whole map should have a visually nice and information-rich background.

As always, Johannes will have to include the right libraries and set up the environment. Johannes is going to use the `mindmap` library and since he wishes to show a calendar, he will also need the `calendar` library. In order to put something on a background layer, it seems like a good idea to also include the `backgrounds` library.

6.2 Introduction to Trees

The first choice Johannes must make is whether he will organize the concepts as a tree, with root concepts and concept branches and leaf concepts, or as a general graph. The tree implicitly organizes the concepts, while a graph is more flexible. Johannes decides to compromise: Basically, the concepts will be organized as a tree. However, he will selectively add connections between concepts that are related, but which appear on different levels or branches of the tree.

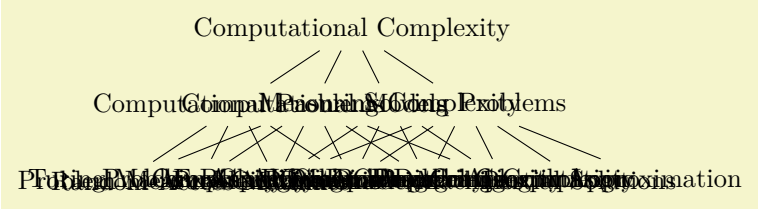
Johannes starts with a tree-like list of concepts that he feels are important in Computational Complexity:

- Computational Problems
  - Problem Measures
  - Problem Aspects
  - Problem Domains
  - Key Problems
- Computational Models
  - Turing Machines
  - Random-Access Machines
  - Circuits
  - Binary Decision Diagrams
  - Oracle Machines
  - Programming in Logic
- Measuring Complexity
  - Complexity Measures
  - Classifying Complexity
  - Comparing Complexity
  - Describing Complexity
- Solving Problems

- Exact Algorithms
- Randomization
- Fixed-Parameter Algorithms
- Parallel Computation
- Partial Solutions
- Approximation

Johannes will surely need to modify this list later on, but it looks good as a first approximation. He will also need to add a number of subtopics (like *lots* of complexity classes under the topic "classifying complexity"), but he will do this as he constructs the map.

Turning the list of topics into a Ti*k*Z-tree is easy, in principle. The basic idea is that a node can have *children*, which in turn can have children of their own, and so on. To add a child to a node, Johannes can simply write `child {`⟨*node*⟩`}` right after a node. The ⟨*node*⟩ should, in turn, be the code for creating a node. To add another node, Johannes can use `child` once more, and so on. Johannes is eager to try out this construct and writes down the following:




```
\tikz
  \node {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem Measures} }
      child { node {Problem Aspects} }
      child { node {Problem Domains} }
      child { node {Key Problems} }
    }
    child { node {Computational Models}
      child { node {Turing Machines} }
      child { node {Random-Access Machines} }
      child { node {Circuits} }
      child { node {Binary Decision Diagrams} }
      child { node {Oracle Machines} }
      child { node {Programming in Logic} }
    }
    child { node {Measuring Complexity}
      child { node {Complexity Measures} }
      child { node {Classifying Complexity} }
      child { node {Comparing Complexity} }
      child { node {Describing Complexity} }
    }
    child { node {Solving Problems}
      child { node {Exact Algorithms} }
      child { node {Randomization} }
      child { node {Fixed-Parameter Algorithms} }
      child { node {Parallel Computation} }
      child { node {Partial Solutions} }
      child { node {Approximation} }
    };
```

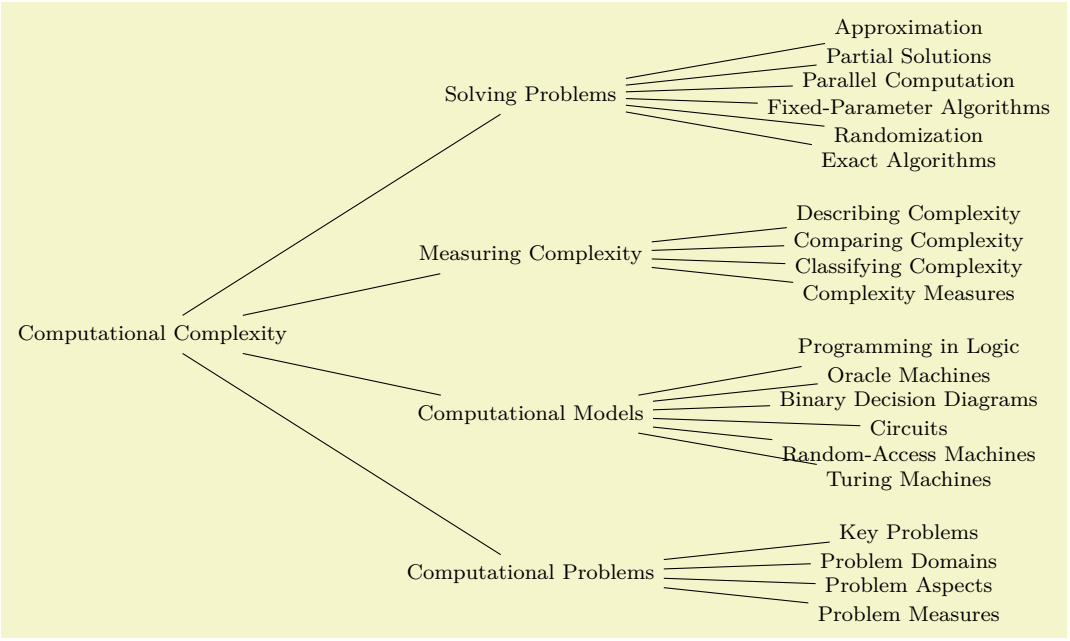
Well, that did not quite work out as expected (although, what, exactly, did one expect?). There are two problems:

1. The overlap of the nodes is due to the fact that Ti*k*Z is not particularly smart when it comes to placing child nodes. Even though it is possible to configure Ti*k*Z to use rather clever placement methods, Ti*k*Z has no way of taking the actual size of the child nodes into account. This may seem strange but the reason is that the child nodes are rendered and placed one at a time, so the size of the last node is not known when the first node is being processed. In essence, you have to specify appropriate level and sibling node spacings "by hand".

2. The standard computer-science-top-down rendering of a tree is rather ill-suited to visualizing the concepts. It would be better to either rotate the map by ninety degrees or, even better, to use some

sort of circular arrangement.

Johannes redraws the tree, but this time with some more appropriate options set, which he found more or less by trial-and-error:



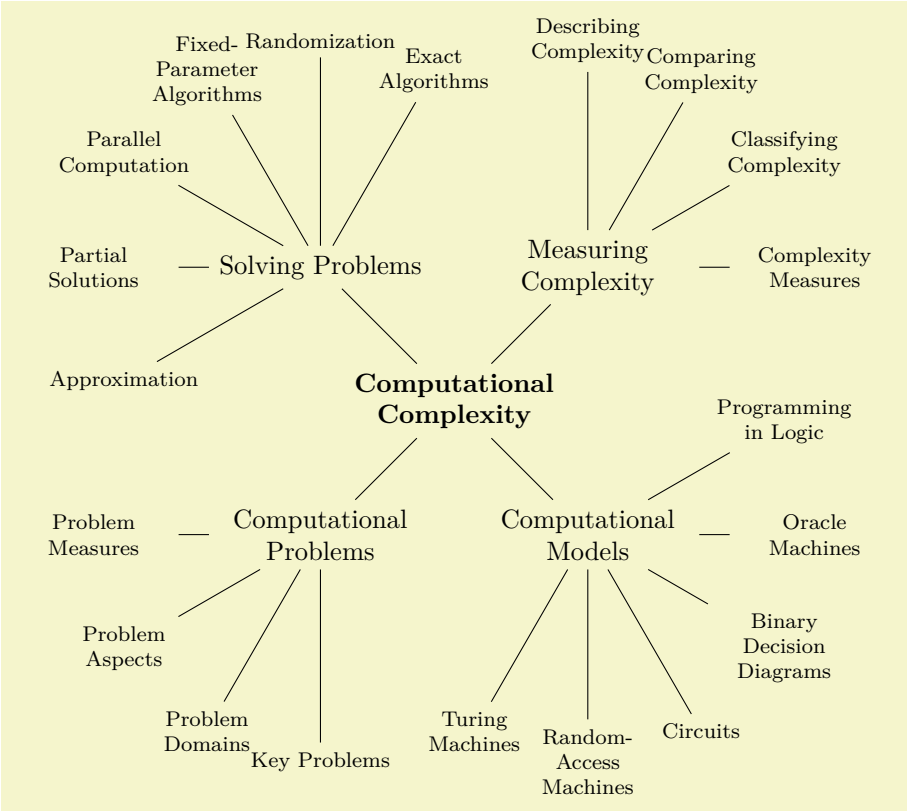

```
\usetikzlibrary {trees}
\tikz [font=\footnotesize,
       grow=right, level 1/.style={sibling distance=6em},
                   level 2/.style={sibling distance=1em}, level distance=5cm]
  \node {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem Measures} }
      child { node {Problem Aspects} }
      ... % as before
```

Still not quite what Johannes had in mind, but he is getting somewhere.

For configuring the tree, two parameters are of particular importance: The `level distance` tells Ti*k*Z the distance between (the centers of) the nodes on adjacent levels or layers of a tree. The `sibling distance` is, as the name suggests, the distance between (the centers of) siblings of the tree.

You can globally set these parameters for a tree by simply setting them somewhere before the tree starts, but you will typically wish them to be different for different levels of the tree. In this case, you should set styles like `level 1` or `level 2`. For the first level of the tree, the `level 1` style is used, for the second level the `level 2` style, and so on. You can also set the sibling and level distances only for certain nodes by passing these options to the `child` command as options. (Note that the options of a `node` command are local to the node and have no effect on the children. Also note that it is possible to specify options that do have an effect on the children. Finally note that specifying options for children "at the right place" is an arcane art and you should peruse Section 21.4 on a rainy Sunday afternoon, if you are really interested.)

The `grow` key is used to configure the direction in which a tree grows. You can change growth direction "in the middle of a tree" simply by changing this key for a single child or a whole level. By including the `trees` library you also get access to additional growth strategies such as a "circular" growth:




```
\usetikzlibrary {trees}
\tikz [text width=2.7cm, align=flush center,
       grow cyclic,
       level 1/.style={level distance=2.5cm,sibling angle=90},
       level 2/.style={text width=2cm, font=\footnotesize, level distance=3cm,sibling angle=30}]
  \node[font=\bfseries] {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem Measures} }
      child { node {Problem Aspects} }
      ... % as before
```

Johannes is pleased to learn that he can access and manipulate the nodes of the tree like any normal node. In particular, he can name them using the `name=` option or the `(⟨name⟩)` notation and he can use any available shape or style for the trees nodes. He can connect trees later on using the normal `\draw (some node) -- (another node);` syntax. In essence, the `child` command just computes an appropriate position for a node and adds a line from the child to the parent node.

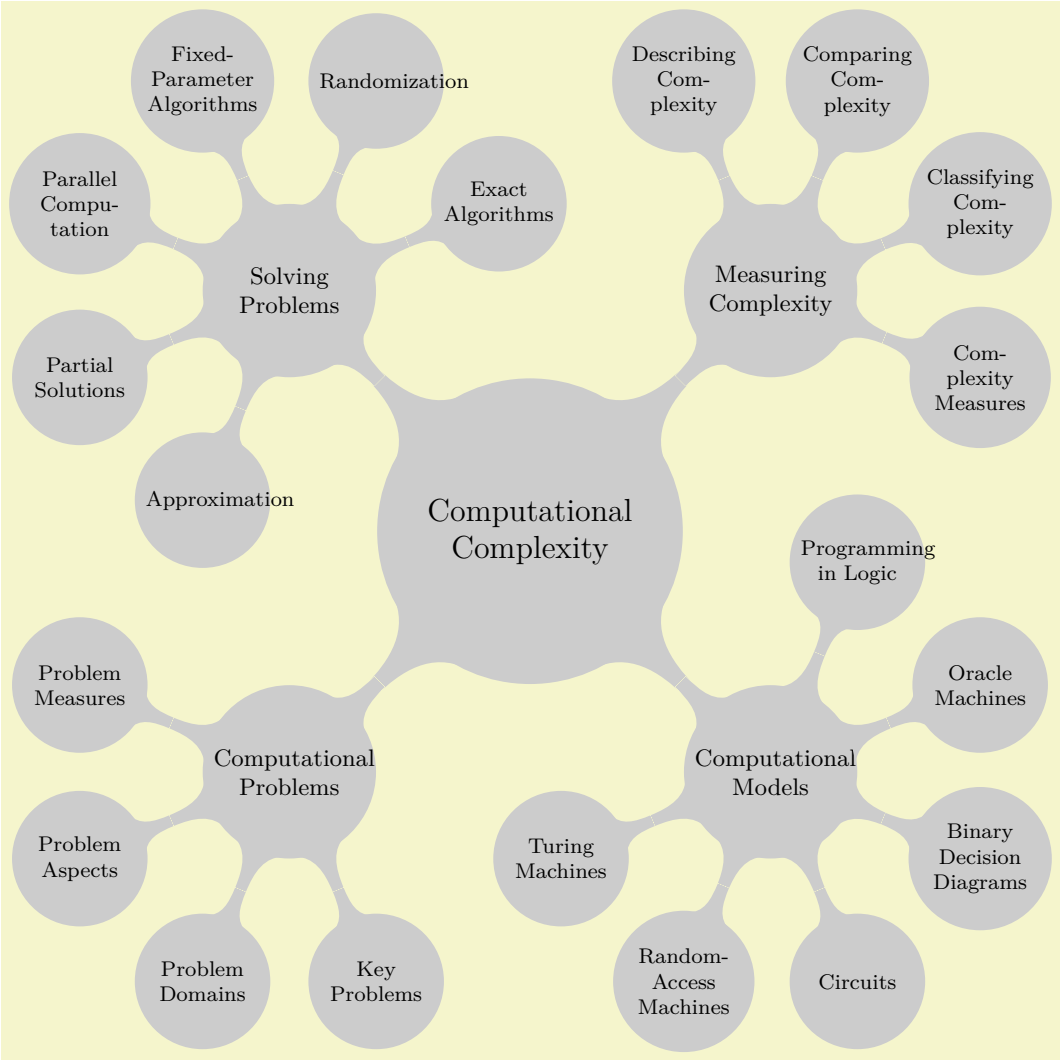
6.3 Creating the Lecture Map

Johannes now has a first possible layout for his lecture map. The next step is to make it "look nicer". For this, the `mindmap` library is helpful since it makes a number of styles available that will make a tree look like a nice "mind map" or "concept map".

The first step is to include the `mindmap` library, which Johannes already did. Next, he must add one of the following options to a scope that will contain the lecture map: `mindmap` or `large mindmap` or `huge mindmap`. These options all have the same effect, except that for a `large mindmap` the predefined font size and node sizes are somewhat larger than for a standard `mindmap` and for a `huge mindmap` they are even larger. So, a `large mindmap` does not necessarily need to have a lot of concepts, but it will need a lot of paper.

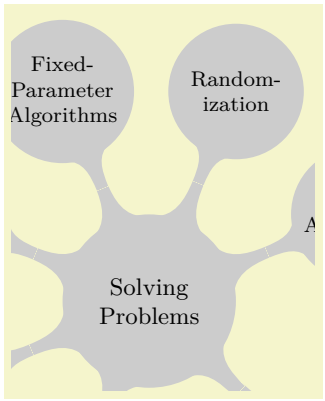
The second step is to add the `concept` option to every node that will, indeed, be a concept of the mindmap. The idea is that some nodes of a tree will be real concepts, while other nodes might just be "simple children". Typically, this is not the case, so you might consider saying `every node/.style=concept`.

The third step is to set up the sibling *angle* (rather than a sibling distance) to specify the angle between sibling concepts.




```
\usetikzlibrary {mindmap}
\tikz [mindmap, every node/.style=concept, concept color=black!20,
       grow cyclic,
       level 1/.append style={level distance=4.5cm,sibling angle=90},
       level 2/.append style={level distance=3cm,sibling angle=45}]
  \node [root concept] {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem Measures} }
      child { node {Problem Aspects} }
      ... % as before
```

When Johannes typesets the above map, TEX (rightfully) starts complaining about several overfull boxes and, indeed, words like “Randomization” stretch out beyond the circle of the concept. This seems a bit mysterious at first sight: Why does TEX not hyphenate the word? The reason is that TEX will never hyphenate the first word of a paragraph because it starts looking for “hyphenatable” letters only after a so-called glue. In order to have TEX hyphenate these single words, Johannes must use a bit of evil trickery: He inserts a `\hskip0pt` before the word. This has no effect except for inserting an (invisible) glue before the word and, thereby, allowing TEX to hyphenate the first word also. Since Johannes does not want to add `\hskip0pt` inside each node, he uses the `execute at begin node` option to make Ti*k*Z insert this text with every node.



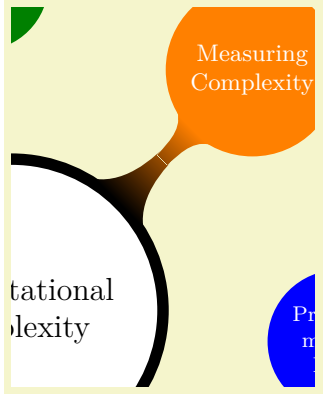

```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [mindmap,
   every node/.style={concept, execute at begin node=\hskip0pt},
   concept color=black!20,
   grow cyclic,
   level 1/.append style={level distance=4.5cm,sibling angle=90},
   level 2/.append style={level distance=3cm,sibling angle=45}]
  \clip (-1,2) rectangle ++ (-4,5);
  \node [root concept] {Computational Complexity} % root
    child { node {Computational Problems}
      child { node {Problem Measures} }
      child { node {Problem Aspects} }
      ... % as before
\end{tikzpicture}
```

In the above example a clipping was used to show only part of the lecture map, in order to save space. The same will be done in the following examples, we return to the complete lecture map at the end of this tutorial.

Johannes is now eager to colorize the map. The idea is to use different colors for different parts of the map. He can then, during his lectures, talk about the "green" or the "red" topics. This will make it easier for his students to locate the topic he is talking about on the map. Since "computational problems" somehow sounds "problematic", Johannes chooses red for them, while he picks green for the "solving problems". The topics "measuring complexity" and "computational models" get more neutral colors; Johannes picks orange and blue.

To set the colors, Johannes must use the `concept color` option, rather than just, say, `node [fill=red]`. Setting just the fill color to `red` would, indeed, make the node red, but it would *just* make the node red and not the bar connecting the concept to its parent and also not its children. By comparison, the special `concept color` option will not only set the color of the node and its children, but it will also (magically) create appropriate shadings so that the color of a parent concept smoothly changes to the color of a child concept.

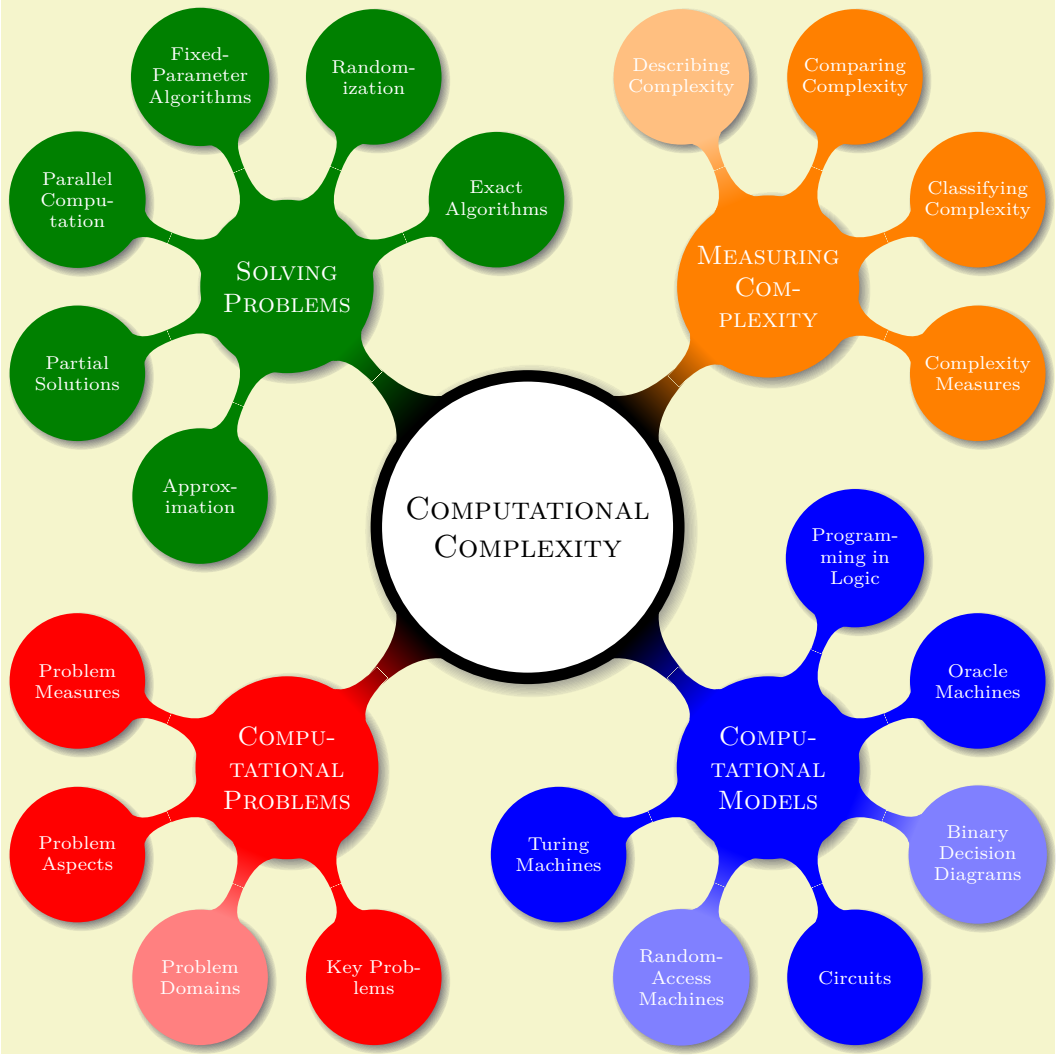
For the root concept Johannes decides to do something special: He sets the concept color to black, sets the line width to a large value, and sets the fill color to white. The effect of this is that the root concept will be encircled with a thick black line and the children are connected to the central concept via bars.




```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [mindmap,
   every node/.style={concept, execute at begin node=\hskip0pt},
   root concept/.append style={
     concept color=black, fill=white, line width=1ex, text=black},
   text=white,
   grow cyclic,
   level 1/.append style={level distance=4.5cm,sibling angle=90},
   level 2/.append style={level distance=3cm,sibling angle=45}]
   \clip (0,-1) rectangle ++(4,5);
  \node [root concept] {Computational Complexity} % root
    child [concept color=red] { node {Computational Problems}
      child { node {Problem Measures} }
      ... % as before
    }
    child [concept color=blue] { node {Computational Models}
      child { node {Turing Machines} }
      ... % as before
    }
    child [concept color=orange] { node {Measuring Complexity}
      child { node {Complexity Measures} }
      ... % as before
    }
    child [concept color=green!50!black] { node {Solving Problems}
      child { node {Exact Algorithms} }
      ... % as before
    };
\end{tikzpicture}
```

Johannes adds three finishing touches: First, he changes the font of the main concepts to small caps. Second, he decides that some concepts should be "faded", namely those that are important in principle

and belong on the map, but which he will not talk about in his lecture. To achieve this, Johannes defines four styles, one for each of the four main branches. These styles (a) set up the correct concept color for the whole branch and (b) define the `faded` style appropriately for this branch. Third, he adds a `circular drop shadow`, defined in the `shadows` library, to the concepts, just to make things look a bit more fancy.



```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap]
  \begin{scope}[
    every node/.style={concept, circular drop shadow,execute at begin node=\hskip0pt},
    root concept/.append style={
      concept color=black, fill=white, line width=1ex, text=black, font=\large\scshape},
    text=white,
    computational problems/.style={concept color=red,faded/.style={concept color=red!50}},
    computational models/.style={concept color=blue,faded/.style={concept color=blue!50}},
    measuring complexity/.style={concept color=orange,faded/.style={concept color=orange!50}},
    solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}},
    grow cyclic,
    level 1/.append style={level distance=4.5cm,sibling angle=90,font=\scshape},
    level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}]
    \node [root concept] {Computational Complexity} % root
      child [computational problems] { node {Computational Problems}
        child         { node {Problem Measures} }
        child         { node {Problem Aspects} }
        child [faded] { node {Problem Domains} }
        child         { node {Key Problems} }
      }
      child [computational models] { node {Computational Models}
        child         { node {Turing Machines} }
        child [faded] { node {Random-Access Machines} }
        ...
  \end{scope}
\end{tikzpicture}
```

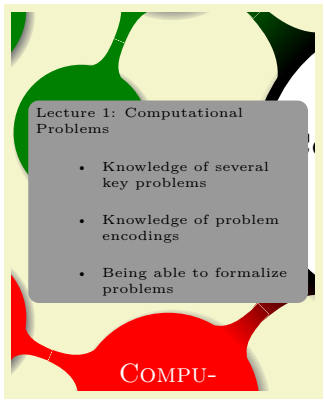
6.4 Adding the Lecture Annotations

Johannes will give about a dozen lectures during the course "computational complexity". For each lecture he has compiled a (short) list of learning targets that state what knowledge and qualifications his students should acquire during this particular lecture (note that learning targets are not the same as the contents of a lecture). For each lecture he intends to put a little rectangle on the map containing these learning targets and the name of the lecture, each time somewhere near the topic of the lecture. Such "little rectangles" are called "annotations" by the `mindmap` library.

In order to place the annotations next to the concepts, Johannes must assign names to the nodes of the concepts. He could rely on Ti*k*Z's automatic naming of the nodes in a tree, where the children of a node named `root` are named `root-1`, `root-2`, `root-3`, and so on. However, since Johannes is not sure about the final order of the concepts in the tree, it seems better to explicitly name all concepts of the tree in the following manner:

```
\node [root concept] (Computational Complexity) {Computational Complexity}
  child [computational problems] { node (Computational Problems) {Computational Problems}
    child         { node (Problem Measures) {Problem Measures} }
    child         { node (Problem Aspects) {Problem Aspects} }
    child [faded] { node (Problem Domains) {Problem Domains} }
    child         { node (Key Problems) {Key Problems} }
  }
...
```

The `annotation` style of the `mindmap` library mainly sets up a rectangular shape of appropriate size. Johannes configures the style by defining `every annotation` appropriately.

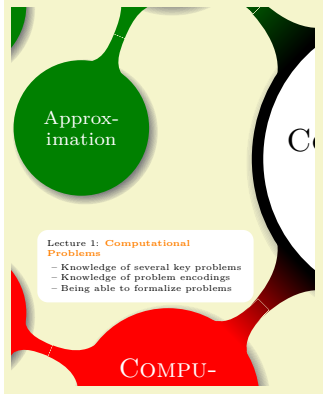



```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap]
  \clip (-5,-5) rectangle ++ (4,5);
  \begin{scope}[
     every node/.style={concept, circular drop shadow, ...}] % as before
    \node [root concept] (Computational Complexity)    ...   % as before
  \end{scope}

  \begin{scope}[every annotation/.style={fill=black!40}]
    \node [annotation, above] at (Computational Problems.north) {
      Lecture 1: Computational Problems
      \begin{itemize}
      \item Knowledge of several key problems
      \item Knowledge of problem encodings
      \item Being able to formalize problems
      \end{itemize}
    };
  \end{scope}
\end{tikzpicture}
```

Well, that does not yet look quite perfect. The spacing or the `{itemize}` is not really appropriate and the node is too large. Johannes can configure these things "by hand", but it seems like a good idea to define a macro that will take care of these things for him. The "right" way to do this is to define a `\lecture` macro that takes a list of key–value pairs as argument and produces the desired annotation. However, to keep things simple, Johannes' `\lecture` macro simply takes a fixed number of arguments having the following meaning: The first argument is the number of the lecture, the second is the name of the lecture, the third are positioning options like `above`, the fourth is the position where the node is placed, the fifth is the list of items to be shown, and the sixth is a date when the lecture will be held (this parameter is not yet needed, we will, however, need it later on).

```
\def\lecture#1#2#3#4#5#6{
  \node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm] at (#4) {
    Lecture #1: \textcolor{orange}{\textbf{#2}}
    \list{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
              \parskip=0pt\labelwidth=8pt\leftmargin=8pt
              \itemindent=0pt\labelsep=2pt}
    #5
    \endlist
  };
}
```




```
\usetikzlibrary {mindmap,shadows}
\begin{tikzpicture}[mindmap,every annotation/.style={fill=white}]
  \clip (-5,-5) rectangle ++ (4,5);
  \begin{scope}[
     every node/.style={concept, circular drop shadow, ... % as before
    \node [root concept] (Computational Complexity)    ... % as before
  \end{scope}

  \lecture{1}{Computational Problems}{above,xshift=-3mm}
  {Computational Problems.north}{
    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
  }{2009-04-08}
\end{tikzpicture}
```

In the same fashion Johannes can now add the other lecture annotations. Obviously, Johannes will have some trouble fitting everything on a single A4-sized page, but by adjusting the spacing and some experimentation he can quickly arrange all the annotations as needed.

6.5 Adding the Background

Johannes has already used colors to organize his lecture map into four regions, each having a different color. In order to emphasize these regions even more strongly, he wishes to add a background coloring to each of these regions.

Adding these background colors turns out to be more tricky than Johannes would have thought. At first sight, what he needs is some sort of "color wheel" that is blue in the lower right direction and then

changes smoothly to orange in the upper right direction and then to green in the upper left direction and so on. Unfortunately, there is no easy way of creating such a color wheel shading (although it can be done, in principle, but only at a very high cost, see page 777 for an example).

Johannes decides to do something a bit more basic: He creates four large rectangles, one for each of the four quadrants around the central concept, each colored with a light version of the quadrant. Then, in order to "smooth" the change between adjacent rectangles, he puts four shadings on top of them.

Since these background rectangles should go "behind" everything else, Johannes puts all his background stuff on the `background` layer.

In the following code, only the central concept is shown to save some space:




```
\usetikzlibrary {backgrounds,mindmap,shadows}
\begin{tikzpicture}[
  mindmap,
  concept color=black,
  root concept/.append style={
    concept,
    circular drop shadow,
    fill=white, line width=1ex,
    text=black, font=\large\scshape}
  ]

  \clip (-1.5,-5) rectangle ++(4,10);

  \node [root concept] (Computational Complexity) {Computational Complexity};

  \begin{pgfonlayer}{background}
    \clip (-1.5,-5) rectangle ++(4,10);

    \colorlet{upperleft}{green!50!black!25}
    \colorlet{upperright}{orange!25}
    \colorlet{lowerleft}{red!25}
    \colorlet{lowerright}{blue!25}

     % The large rectangles:
    \fill [upperleft]  (Computational Complexity) rectangle ++(-20,20);
    \fill [upperright] (Computational Complexity) rectangle ++(20,20);
    \fill [lowerleft]  (Computational Complexity) rectangle ++(-20,-20);
    \fill [lowerright] (Computational Complexity) rectangle ++(20,-20);

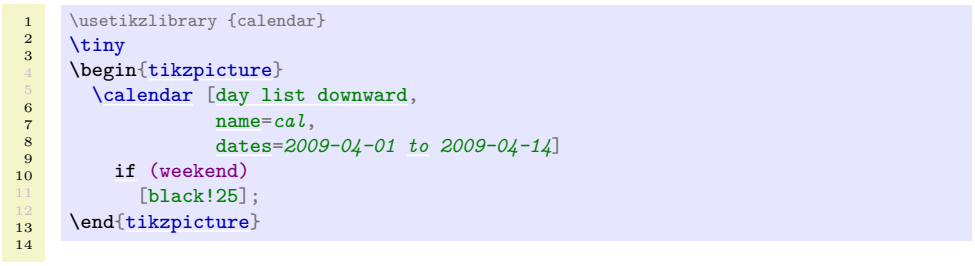
    % The shadings:
    \shade [left color=upperleft,right color=upperright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
    \shade [left color=lowerleft,right color=lowerright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
    \shade [top color=upperleft,bottom color=lowerleft]
      ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
    \shade [top color=upperright,bottom color=lowerright]
      ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);
  \end{pgfonlayer}
\end{tikzpicture}
```

6.6 Adding the Calendar

Johannes intends to plan his lecture rather carefully. In particular, he already knows when each of his lectures will be held during the course. Naturally, this does not mean that Johannes will slavishly follow the plan and he might need longer for some subjects than he anticipated, but nevertheless he has a detailed plan of when which subject will be addressed.

Johannes intends to share this plan with his students by adding a calendar to the lecture map. In addition to serving as a reference on which particular day a certain topic will be addressed, the calendar is also useful to show the overall chronological order of the course.

In order to add a calendar to a TikZ graphic, the `calendar` library is most useful. The library provides the `\calendar` command, which takes a large number of options and which can be configured in many ways to produce just about any kind of calendar imaginable. For Johannes' purposes, a simple `day list downward` will be a nice option since it produces a list of days that go "downward".



```
\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
  \calendar [day list downward,
             name=cal,
             dates=2009-04-01 to 2009-04-14]
    if (weekend)
      [black!25];
\end{tikzpicture}
```

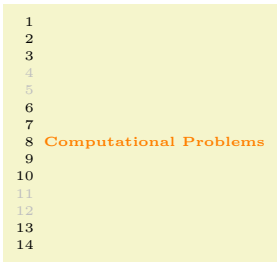
Using the `name` option, we gave a name to the calendar, which will allow us to reference the nodes that make up the individual days of the calendar later on. For instance, the rectangular node containing the `1` that represents April 1st, 2009, can be referenced as `(cal-2009-04-01)`. The `dates` option is used to specify an interval for which the calendar should be drawn. Johannes will need several months in his calendar, but the above example only shows two weeks to save some space.

Note the `if (weekend)` construct. The `\calendar` command is followed by options and then by `if`-statements. These `if`-statements are checked for each day of the calendar and when a date passes this test, the options or the code following the `if`-statement is executed. In the above example, we make weekend days (Saturdays and Sundays, to be precise) lighter than normal days. (Use your favorite calendar to check that, indeed, April 5th, 2009, is a Sunday.)

As mentioned above, Johannes can reference the nodes that are used to typeset days. Recall that his `\lecture` macro already got passed a date, which we did not use, yet. We can now use it to place the lecture's title next to the date when the lecture will be held:

```
\def\lecture#1#2#3#4#5#6{
  % As before:
  \node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm] at (#4) {
    Lecture #1: \textcolor{orange}{\textbf{#2}}
    \list{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
              \parskip=0pt\labelwidth=8pt\leftmargin=8pt
              \itemindent=0pt\labelsep=2pt}
    #5
    \endlist
  };
  % New:
  \node [anchor=base west] at (cal-#6.base east) {\textcolor{orange}{\textbf{#2}}};
}
```

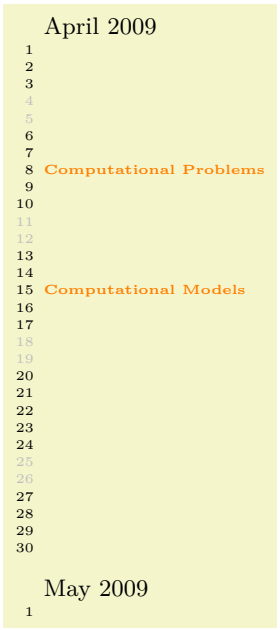
Johannes can now use this new `\lecture` command as follows (in the example, only the new part of the definition is used):




```
\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
  \calendar [day list downward,
             name=cal,
             dates=2009-04-01 to 2009-04-14]
    if (weekend)
      [black!25];

  % As before:
  \lecture{1}{Computational Problems}{above,xshift=-3mm}
  {Computational Problems.north}{
    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
  }{2009-04-08}
\end{tikzpicture}
```

As a final step, Johannes needs to add a few more options to the calendar command: He uses the `month text` option to configure how the text of a month is rendered (see Section 47 for details) and then typesets the month text at a special position at the beginning of each month.




```
\usetikzlibrary {calendar}
\tiny
\begin{tikzpicture}
  \calendar [day list downward,
             month text=\%mt\ \%y0,
             month yshift=3.5em,
             name=cal,
             dates=2009-04-01 to 2009-05-01]
    if (weekend)
      [black!25]
    if (day of month=1) {
      \node at (0pt,1.5em) [anchor=base west] {\small\tikzmonthtext};
    };

  \lecture{1}{Computational Problems}{above,xshift=-3mm}
  {Computational Problems.north}{
    \item Knowledge of several key problems
    \item Knowledge of problem encodings
    \item Being able to formalize problems
  }{2009-04-08}

  \lecture{2}{Computational Models}{above,xshift=-3mm}
  {Computational Models.north}{
    \item Knowledge of Turing machines
    \item Being able to compare the computational power of different
      models
  }{2009-04-15}
\end{tikzpicture}
```

6.7 The Complete Code

Putting it all together, Johannes gets the following code:

First comes the definition of the `\lecture` command:

```
\def\lecture#1#2#3#4#5#6{
  % As before:
  \node [annotation, #3, scale=0.65, text width=4cm, inner sep=2mm, fill=white] at (#4) {
    Lecture #1: \textcolor{orange}{\textbf{#2}}
    \list{--}{\topsep=2pt\itemsep=0pt\parsep=0pt
              \parskip=0pt\labelwidth=8pt\leftmargin=8pt
              \itemindent=0pt\labelsep=2pt}
    #5
    \endlist
  };
  % New:
  \node [anchor=base west] at (cal-#6.base east) {\textcolor{orange}{\textbf{#2}}};
}
```

This is followed by the main mindmap setup…

```
\noindent
\begin{tikzpicture}
  \begin{scope}[
    mindmap,
    every node/.style={concept, circular drop shadow,execute at begin node=\hskip0pt},
    root concept/.append style={
      concept color=black,
      fill=white, line width=1ex,
      text=black, font=\large\scshape},
    text=white,
    computational problems/.style={concept color=red,faded/.style={concept color=red!50}},
    computational models/.style={concept color=blue,faded/.style={concept color=blue!50}},
    measuring complexity/.style={concept color=orange,faded/.style={concept color=orange!50}},
    solving problems/.style={concept color=green!50!black,faded/.style={concept color=green!50!black!50}},
    grow cyclic,
    level 1/.append style={level distance=4.5cm,sibling angle=90,font=\scshape},
    level 2/.append style={level distance=3cm,sibling angle=45,font=\scriptsize}]
```

…and contents:

```
\node [root concept] (Computational Complexity) {Computational Complexity} % root
    child [computational problems] { node [yshift=-1cm] (Computational Problems) {Computational Problems}
      child         { node (Problem Measures) {Problem Measures} }
      child         { node (Problem Aspects) {Problem Aspects} }
      child [faded] { node (problem Domains) {Problem Domains} }
      child         { node (Key Problems) {Key Problems} }
    }
    child [computational models] { node [yshift=-1cm]  (Computational Models) {Computational Models}
      child         { node (Turing Machines) {Turing Machines} }
      child [faded] { node (Random-Access Machines) {Random-Access Machines} }
      child         { node (Circuits) {Circuits} }
      child [faded] { node (Binary Decision Diagrams) {Binary Decision Diagrams} }
      child         { node (Oracle Machines) {Oracle Machines} }
      child         { node (Programming in Logic) {Programming in Logic} }
    }
    child [measuring complexity] { node [yshift=1cm] (Measuring Complexity) {Measuring Complexity}
      child         { node (Complexity Measures) {Complexity Measures} }
      child         { node (Classifying Complexity) {Classifying Complexity} }
      child         { node (Comparing Complexity) {Comparing Complexity} }
      child [faded] { node (Describing Complexity) {Describing Complexity} }
    }
    child [solving problems] { node [yshift=1cm] (Solving Problems) {Solving Problems}
      child         { node (Exact Algorithms) {Exact Algorithms} }
      child         { node (Randomization) {Randomization} }
      child         { node (Fixed-Parameter Algorithms) {Fixed-Parameter Algorithms} }
      child         { node (Parallel Computation) {Parallel Computation} }
      child         { node (Partial Solutions) {Partial Solutions} }
      child         { node (Approximation) {Approximation} }
    };
\end{scope}
```

Now comes the calendar code:

```
\tiny
\calendar [day list downward,
           month text=\%mt\ \%y0,
           month yshift=3.5em,
           name=cal,
           at={(-.5\textwidth-5mm,.5\textheight-1cm)},
           dates=2009-04-01 to 2009-06-last]
  if (weekend)
    [black!25]
  if (day of month=1) {
    \node at (0pt,1.5em) [anchor=base west] {\small\tikzmonthtext};
  };
```

The lecture annotations:

```
\lecture{1}{Computational Problems}{above,xshift=-5mm,yshift=5mm}{Computational Problems.north}{
  \item Knowledge of several key problems
  \item Knowledge of problem encodings
  \item Being able to formalize problems
}{2009-04-08}

\lecture{2}{Computational Models}{above left}
{Computational Models.west}{
  \item Knowledge of Turing machines
  \item Being able to compare the computational power of different
    models
}{2009-04-15}
```

Finally, the background:

```
  \begin{pgfonlayer}{background}
    \clip[xshift=-1cm] (-.5\textwidth,-.5\textheight) rectangle ++(\textwidth,\textheight);

    \colorlet{upperleft}{green!50!black!25}
    \colorlet{upperright}{orange!25}
    \colorlet{lowerleft}{red!25}
    \colorlet{lowerright}{blue!25}

     % The large rectangles:
    \fill [upperleft]  (Computational Complexity) rectangle ++(-20,20);
    \fill [upperright] (Computational Complexity) rectangle ++(20,20);
    \fill [lowerleft]  (Computational Complexity) rectangle ++(-20,-20);
    \fill [lowerright] (Computational Complexity) rectangle ++(20,-20);

    % The shadings:
    \shade [left color=upperleft,right color=upperright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,20);
    \shade [left color=lowerleft,right color=lowerright]
      ([xshift=-1cm]Computational Complexity) rectangle ++(2,-20);
    \shade [top color=upperleft,bottom color=lowerleft]
      ([yshift=-1cm]Computational Complexity) rectangle ++(-20,2);
    \shade [top color=upperright,bottom color=lowerright]
      ([yshift=-1cm]Computational Complexity) rectangle ++(20,2);
  \end{pgfonlayer}
\end{tikzpicture}
```

The next page shows the resulting lecture map in all its glory (it would be somewhat more glorious, if there were more lecture annotations, but you should get the idea).

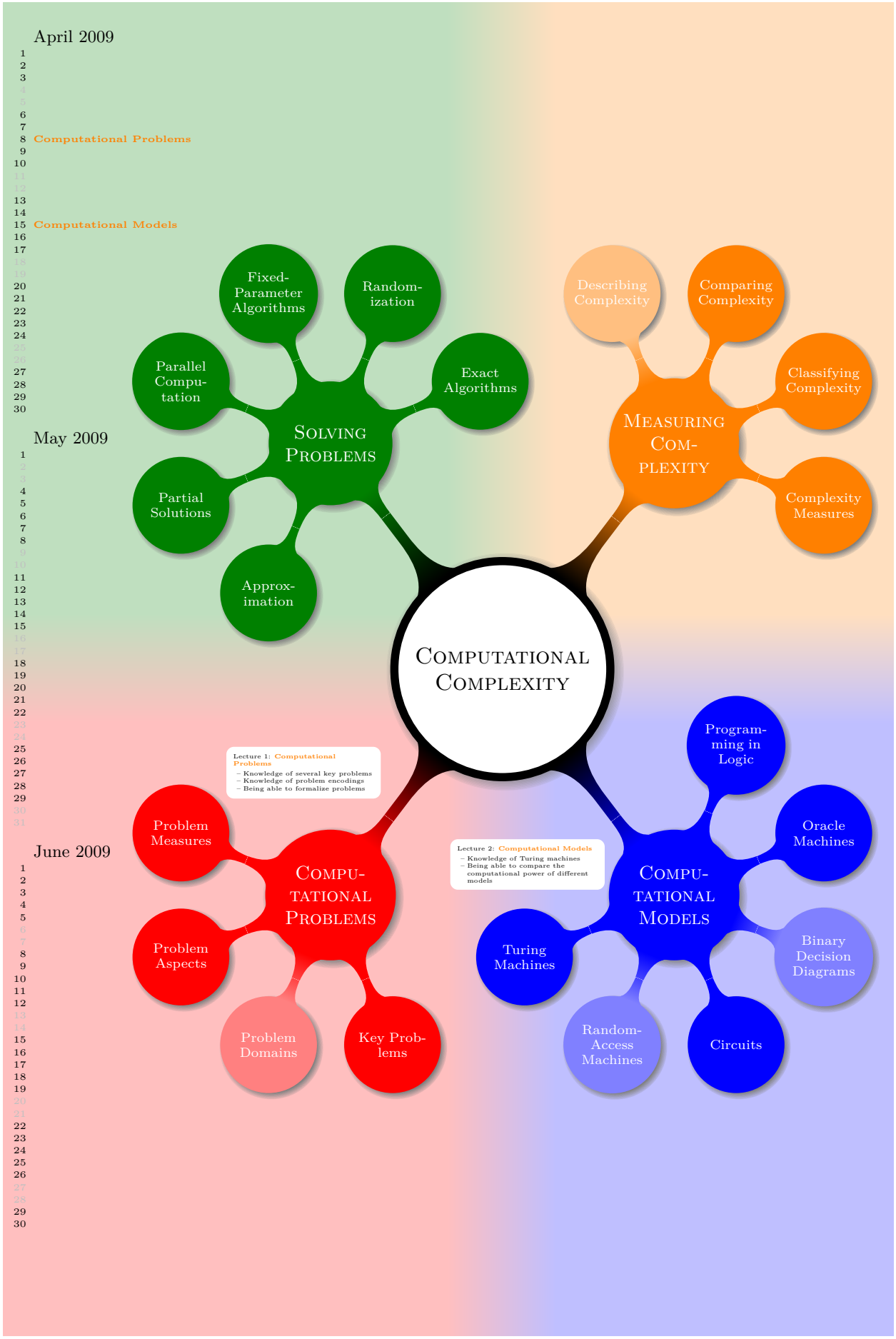
April 2009

Computational Problems

Computational Models

May 2009

June 2009



7 Guidelines on Graphics

The present section is not about PGF or Ti*k*Z, but about general guidelines and principles concerning the creation of graphics for scientific presentations, papers, and books.

The guidelines in this section come from different sources. Many of them are just what I would like to claim is "common sense", some reflect my personal experience (though, hopefully, not my personal preferences), some come from books (the bibliography is still missing, sorry) on graphic design and typography. The most influential source are the brilliant books by Edward Tufte. While I do not agree with everything written in these books, many of Tufte's arguments are so convincing that I decided to repeat them in the following guidelines.

The first thing you should ask yourself when someone presents a bunch of guidelines is: Should I really follow these guidelines? This is an important question, because there are good reasons not to follow general guidelines. The person who set up the guidelines may have had other objectives than you do. For example, a guideline might say "use the color red for emphasis". While this guideline makes perfect sense for, say, a presentation using a projector, red "color" has the *opposite* effect of "emphasis" when printed using a black-and-white printer. Guidelines were almost always set up to address a specific situation. If you are not in this situation, following a guideline can do more harm than good.

The second thing you should be aware of is the basic rule of typography is: "Every rule can be broken, as long as you are *aware* that you are breaking a rule." This rule also applies to graphics. Phrased differently, the basic rule states: "The only mistakes in typography are things done in ignorance." When you are aware of a rule and when you decide that breaking the rule has a desirable effect, break the rule.

7.1 Planning the Time Needed for the Creation of Graphics

When you create a paper with numerous graphics, the time needed to create these graphics becomes an important factor. How much time should you calculate for the creation of graphics?

As a general rule, assume that a graphic will need as much time to create as would a text of the same length. For example, when I write a paper, I need about one hour per page for the first draft. Later, I need between two and four hours per page for revisions. Thus, I expect to need about half an hour for the creation of *a first draft* of a half page graphic. Later on, I expect another one to two hours before the final graphic is finished.

In many publications, even in good journals, the authors and editors have obviously invested a lot of time on the text, but seem to have spend about five minutes to create all of the graphics. Graphics often seem to have been added as an "afterthought" or look like a screen shot of whatever the authors's statistical software shows them. As will be argued later on, the graphics that programs like GNUPLOT produce by default are of poor quality.

Creating informative graphics that help the reader and that fit together with the main text is a difficult, lengthy process.

- Treat graphics as first-class citizens of your papers. They deserve as much time and energy as the text does. Indeed, the creation of graphics might deserve *even more* time than the writing of the main text since more attention will be paid to the graphics and they will be looked at first.
- Plan as much time for the creation and revision of a graphic as you would plan for text of the same size.
- Difficult graphics with a high information density may require even more time.
- Very simple graphics will require less time, but most likely you do not want to have "very simple graphics" in your paper, anyway; just as you would not like to have a "very simple text" of the same size.

7.2 Workflow for Creating a Graphic

When you write a (scientific) paper, you will most likely follow the following pattern: You have some results/ideas that you would like to report about. The creation of the paper will typically start with compiling a rough outline. Then, the different sections are filled with text to create a first draft. This draft is then revised repeatedly until, often after substantial revision, a final paper results. In a good journal paper there is typically not be a single sentence that has survived unmodified from the first draft.

Creating a graphics follows the same pattern:

- Decide on what the graphic should communicate. Make this a conscious decision, that is, determine "What is the graphic supposed to tell the reader?"
- Create an "outline", that is, the rough overall "shape" of the graphic, containing the most crucial elements. Often, it is useful to do this using pencil and paper.
- Fill out the finer details of the graphic to create a first draft.
- Revise the graphic repeatedly along with the rest of the paper.

7.3 Linking Graphics With the Main Text

Graphics can be placed at different places in a text. Either, they can be inlined, meaning they are somewhere "in the middle of the text" or they can be placed in stand-alone "figures". Since printers (the people) like to have their pages "filled", (both for aesthetic and economic reasons) stand-alone figures may traditionally be placed on pages in the document far away from the main text that refers to them. LaTeX and TeX tend to encourage this "drifting away" of graphics for technical reasons.

When a graphic is inlined, it will more or less automatically be linked with the main text in the sense that the labels of the graphic will be implicitly explained by the surrounding text. Also, the main text will typically make it clear what the graphic is about and what is shown.

Quite differently, a stand-alone figure will often be viewed at a time when the main text that this graphic belongs to either has not yet been read or has been read some time ago. For this reason, you should follow the following guidelines when creating stand-alone figures:

- Stand-alone figures should have a caption than should make them "understandable by themselves".

  For example, suppose a graphic shows an example of the different stages of a quicksort algorithm. Then the figure's caption should, at the very least, inform the reader that "the figure shows the different stages of the quicksort algorithm introduced on page xyz". and not just "Quicksort algorithm".
- A good caption adds as much context information as possible. For example, you could say: "The figure shows the different stages of the quicksort algorithm introduced on page xyz. In the first line, the pivot element 5 is chosen. This causes…" While this information can also be given in the main text, putting it in the caption will ensure that the context is kept. Do not feel afraid of a 5-line caption. (Your editor may hate you for this. Consider hating them back.)
- Reference the graphic in your main text as in "for an example of quicksort 'in action', see Figure 2.1 on page xyz".
- Most books on style and typography recommend that you do not use abbreviations as in "Fig. 2.1" but write "Figure 2.1".

  The main argument against abbreviations is that "a period is too valuable to waste it on an abbreviation". The idea is that a period will make the reader assume that the sentence ends after "Fig" and it takes a "conscious backtracking" to realize that the sentence did not end after all.

  The argument in favor of abbreviations is that they save space.

  Personally, I am not really convinced by either argument. On the one hand, I have not yet seen any hard evidence that abbreviations slow readers down. On the other hand, abbreviating all "Figure" by "Fig." is most unlikely to save even a single line in most documents. I avoid abbreviations.

7.4 Consistency Between Graphics and Text

Perhaps the most common "mistake" people do when creating graphics (remember that a "mistake" in design is always just "ignorance") is to have a mismatch between the way their graphics look and the way their text looks.

It is quite common that authors use several different programs for creating the graphics of a paper. An author might produce some plots using GNUPLOT, a diagram using XFIG, and include an `.eps` graphic a coauthor contributed using some unknown program. All these graphics will, most likely, use different line widths, different fonts, and have different sizes. In addition, authors often use options like `[height=5cm]` when including graphics to scale them to some "nice size".

If the same approach were taken to writing the main text, every section would be written in a different font at a different size. In some sections all theorems would be underlined, in another they would be printed

all in uppercase letters, and in another in red. In addition, the margins would be different on each page. Readers and editors would not tolerate a text if it were written in this fashion, but with graphics they often have to.

To create consistency between graphics and text, stick to the following guidelines:

- Do not scale graphics.

  This means that when generating graphics using an external program, create them "at the right size".

- Use the same font(s) both in graphics and the body text.

- Use the same line width in text and graphics.

  The "line width" for normal text is the width of the stem of letters like T. For TEX, this is usually 0.4 pt. However, some journals will not accept graphics with a normal line width below 0.5 pt.

- When using colors, use a consistent color coding in the text and in graphics. For example, if red is supposed to alert the reader to something in the main text, use red also in graphics for important parts of the graphic. If blue is used for structural elements like headlines and section titles, use blue also for structural elements of your graphic.

  However, graphics may also use a logical intrinsic color coding. For example, no matter what colors you normally use, readers will generally assume, say, that the color green as "positive, go, ok" and red as "alert, warning, action".

Creating consistency when using different graphic programs is almost impossible. For this reason, you should consider sticking to a single graphics program.

7.5 Labels in Graphics

Almost all graphics will contain labels, that is, pieces of text that explain parts of the graphics. When placing labels, stick to the following guidelines:

- Follow the rule of consistency when placing labels. You should do so in two ways: First, be consistent with the main text, that is, use the same font as the main text also for labels. Second, be consistent between labels, that is, if you format some labels in some particular way, format all labels in this way.

- In addition to using the same fonts in text and graphics, you should also use the same notation. For example, if you write 1/2 in your main text, also use "1/2" as labels in graphics, not "0.5". A π is a "π" and not "3.141". Finally, $e^{-i\pi}$ is "$e^{-i\pi}$", not "-1", let alone "-1".

- Labels should be legible. They should not only have a reasonably large size, they also should not be obscured by lines or other text. This also applies to labels of lines and text *behind* the labels.

- Labels should be "in place". Whenever there is enough space, labels should be placed next to the thing they label. Only if necessary, add a (subdued) line from the label to the labeled object. Try to avoid labels that only reference explanations in external legends. Reader have to jump back and forth between the explanation and the object that is described.

- Consider subduing "unimportant" labels using, for example, a gray color. This will keep the focus on the actual graphic.

7.6 Plots and Charts

One of the most frequent kind of graphics, especially in scientific papers, are *plots*. They come in a large variety, including simple line plots, parametric plots, three dimensional plots, pie charts, and many more.

Unfortunately, plots are notoriously hard to get right. Partly, the default settings of programs like GNUPLOT or Excel are to blame for this since these programs make it very convenient to create bad plots.

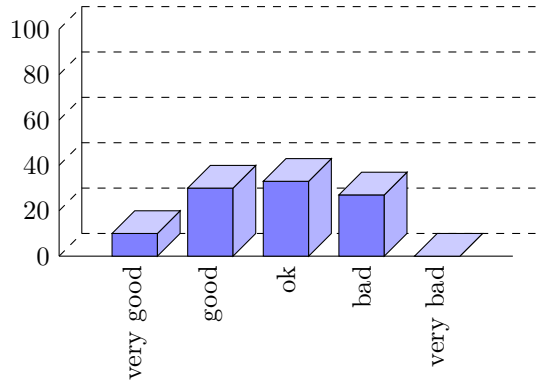
The first question you should ask yourself when creating a plot is: Are there enough data points to merit a plot? If the answer is "not really", use a table.

A typical situation where a plot is unnecessary is when people present a few numbers in a bar diagram. Here is a real-life example: At the end of a seminar a lecturer asked the participants for feedback. Of the 50 participants, 30 returned the feedback form. According to the feedback, three participants considered the seminar "very good", nine considered it "good", ten "ok", eight "bad", and no one thought that the seminar was "very bad".

A simple way of summing up this information is the following table:

Rating given	*Participants (out of 50) who gave this rating*	*Percentage*
"very good"	3	6%
"good"	9	18%
"ok"	10	20%
"bad"	8	16%
"very bad"	0	0%
none	20	40%

What the lecturer did was to visualize the data using a 3D bar diagram. It looked like this (except that in reality the numbers where typeset using some extremely low-resolution bitmap font and were near-unreadable):



Both the table and the "plot" have about the same size. If your first thought is "the graphic looks nicer than the table", try to answer the following questions based on the information in the table or in the graphic:

1. How many participants were there?
2. How many participants returned the feedback form?
3. What percentage of the participants returned the feedback form?
4. How many participants checked "very good"?
5. What percentage out of all participants checked "very good"?
6. Did more than a quarter of the participants check "bad" or "very bad"?
7. What percentage of the participants that returned the form checked "very good"?

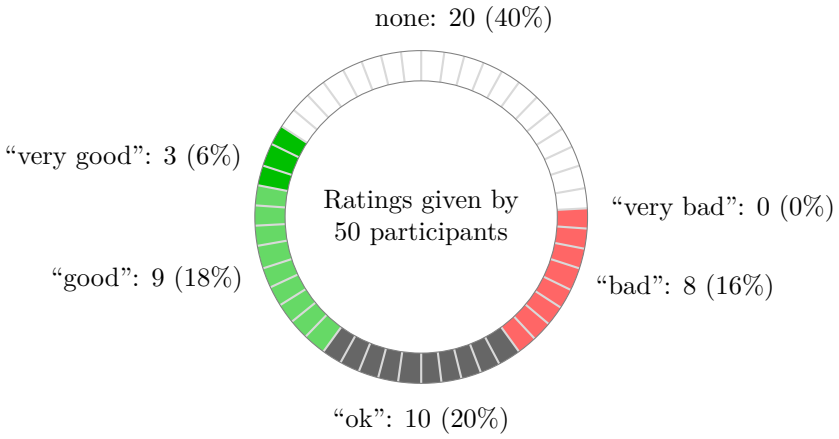
Sadly, the graphic does not allow us to answer *a single one of these questions*. The table answers all of them directly, except for the last one. In essence, the information density of the graphic is very close to zero. The table has a much higher information density; despite the fact that it uses quite a lot of white space to present a few numbers. Here is the list of things that went wrong with the 3D-bar diagram:

- The whole graphic is dominated by irritating background lines.
- It is not clear what the numbers at the left mean; presumably percentages, but it might also be the absolute number of participants.
- The labels at the bottom are rotated, making them hard to read.

  (In the real presentation that I saw, the text was rendered at a very low resolution with about 10 by 6 pixels per letter with wrong kerning, making the rotated text almost impossible to read.)
- The third dimension adds complexity to the graphic without adding information.

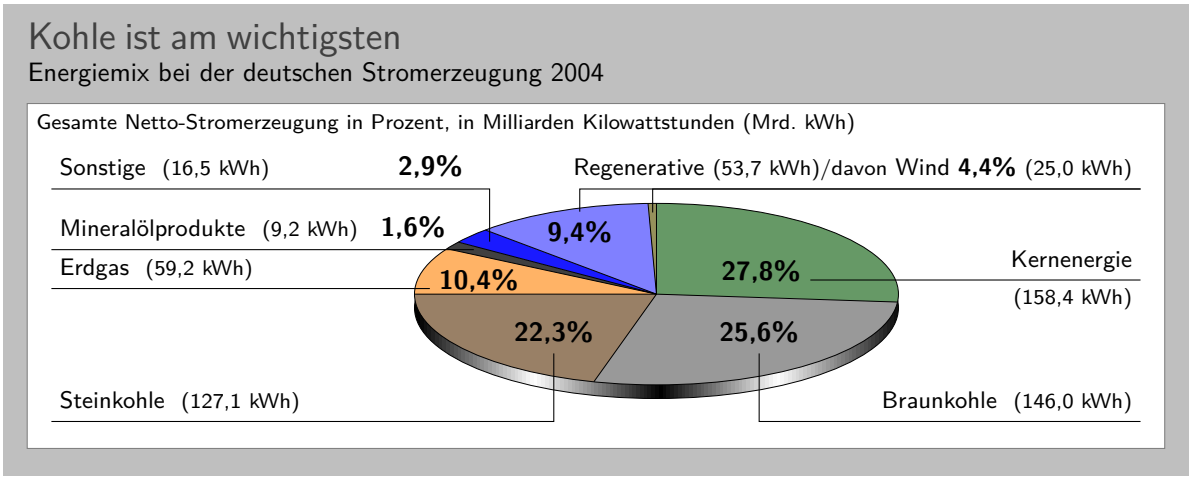
- The three dimensional setup makes it much harder to gauge the height of the bars correctly. Consider the "bad" bar. Is the number this bar stands for more than 20 or less? While the front of the bar is below the 20 line, the back of the bar (which counts) is above.
- It is impossible to tell which numbers are represented by the bars. Thus, the bars needlessly hide the information these bars are all about.
- What do the bar heights add up to? Is it 100% or 60%?
- Does the bar for "very bad" represent 0 or 1?
- Why are the bars blue?

You might argue that in the example the exact numbers are not important for the graphic. The important things is the "message", which is that there are more "very good" and "good" ratings than "bad" and "very bad". However, to convey this message either use a sentence that says so or use a graphic that conveys this message more clearly:



The above graphic has about the same information density as the table (about the same size and the same numbers are shown). In addition, one can directly "see" that there are more good or very good ratings than bad ones. One can also "see" that the number of people who gave no rating at all is not negligible, which is quite common for feedback forms.

Charts are not always a good idea. Let us look at an example that I redrew from a pie chart in *Die Zeit*, June 4th, 2005:



This graphic has been redrawn in Ti*k*Z, but the original looks almost exactly the same.
At first sight, the graphic looks "nice and informative", but there are a lot of things that went wrong:

- The chart is three dimensional. However, the shadings add nothing "information-wise", at best, they distract.

- In a 3D-pie-chart the relative sizes are very strongly distorted. For example, the area taken up by the gray color of "Braunkohle" is larger than the area taken up by the green color of "Kernenergie" *despite the fact that the percentage of Braunkohle is less than the percentage of Kernenergie.*

- The 3D-distortion gets worse for small areas. The area of "Regenerative" somewhat larger than the area of "Erdgas". The area of "Wind" is slightly smaller than the area of "Mineralölprodukte" *although the percentage of Wind is nearly three times larger than the percentage of Mineralölprodukte.*

  In the last case, the different sizes are only partly due to distortion. The designer(s) of the original graphic have also made the "Wind" slice too small, even taking distortion into account. (Just compare the size of "Wind" to "Regenerative" in general.)

- According to its caption, this chart is supposed to inform us that coal was the most important energy source in Germany in 2004. Ignoring the strong distortions caused by the superfluous and misleading 3D-setup, it takes quite a while for this message to get across.

  Coal as an energy source is split up into two slices: one for "Steinkohle" and one for "Braunkohle" (two different kinds of coal). When you add them up, you see that the whole lower half of the pie chart is taken up by coal.

  The two areas for the different kinds of coal are not visually linked at all. Rather, two different colors are used, the labels are on different sides of the graphic. By comparison, "Regenerative" and "Wind" are very closely linked.

- The color coding of the graphic follows no logical pattern at all. Why is nuclear energy green? Regenerative energy is light blue, "other sources" are blue. It seems more like a joke that the area for "Braunkohle" (which literally translates to "brown coal") is stone gray, while the area for "Steinkohle" (which literally translates to "stone coal") is brown.

- The area with the lightest color is used for "Erdgas". This area stands out most because of the brighter color. However, for this chart "Erdgas" is not really important at all.

Edward Tufte calls graphics like the above "chart junk". (I am happy to announce, however, that *Die Zeit* has stopped using 3D pie charts and their information graphics have got somewhat better.)

Here are a few recommendations that may help you avoid producing chart junk:

- Do not use 3D pie charts. They are *evil*.

- Consider using a table instead of a pie chart.

- Do not apply colors randomly; use them to direct the readers's focus and to group things.

- Do not use background patterns, like a crosshatch or diagonal lines, instead of colors. They distract. Background patterns in information graphics are *evil*.

7.7 Attention and Distraction

Pick up your favorite fiction novel and have a look at a typical page. You will notice that the page is very uniform. Nothing is there to distract the reader while reading; no large headlines, no bold text, no large white areas. Indeed, even when the author does wish to emphasize something, this is done using italic letters. Such letters blend nicely with the main text – at a distance you will not be able to tell whether a page contains italic letters, but you would notice a single bold word immediately. The reason novels are typeset this way is the following paradigm: Avoid distractions.

Good typography (like good organization) is something you do *not* notice. The job of typography is to make reading the text, that is, "absorbing" its information content, as effortless as possible. For a novel, readers absorb the content by reading the text line-by-line, as if they were listening to someone telling the story. In this situation anything on the page that distracts the eye from going quickly and evenly from line to line will make the text harder to read.

Now, pick up your favorite weekly magazine or newspaper and have a look at a typical page. You will notice that there is quite a lot "going on" on the page. Fonts are used at different sizes and in different arrangements, the text is organized in narrow columns, typically interleaved with pictures. The reason magazines are typeset in this way is another paradigm: Steer attention.

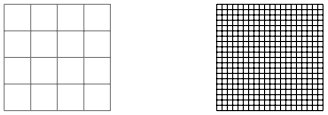
Readers will not read a magazine like a novel. Instead of reading a magazine line-by-line, we use headlines and short abstracts to check whether we want to read a certain article or not. The job of typography is to

steer our attention to these abstracts and headlines, first. Once we have decided that we want to read an article, however, we no longer tolerate distractions, which is why the main text of articles is typeset exactly the same way as a novel.

The two principles "avoid distractions" and "steer attention" also apply to graphics. When you design a graphic, you should eliminate everything that will "distract the eye". At the same time, you should try to actively help the reader "through the graphic" by using fonts/colors/line widths to highlight different parts.

Here is a non-exhaustive list of things that can distract readers:

- Strong contrasts will always be registered first by the eye. For example, consider the following two grids:

  Even though the left grid comes first in English reading order, the right one is much more likely to be seen first: The white-to-black contrast is higher than the gray-to-white contrast. In addition, there are more "places" adding to the overall contrast in the right grid.

  Things like grids and, more generally, help lines usually should not grab the attention of the readers and, hence, should be typeset with a low contrast to the background. Also, a loosely-spaced grid is less distracting than a very closely-spaced grid.

- Dashed lines create many points at which there is black-to-white contrast. Dashed or dotted lines can be very distracting and, hence, should be avoided in general.

  Do not use different dashing patterns to differentiate curves in plots. You lose data points this way and the eye is not particularly good at "grouping things according to a dashing pattern". The eye is *much* better at grouping things according to colors.

- Background patterns filling an area using diagonal lines or horizontal and vertical lines or just dots are almost always distracting and, usually, serve no real purpose.

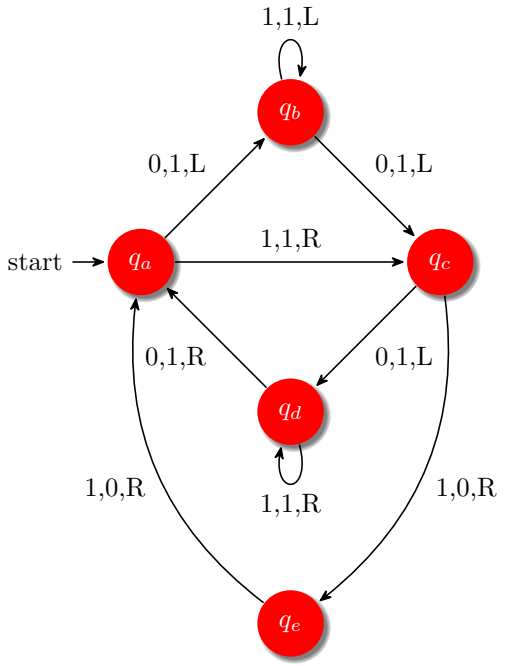
- Background images and shadings distract and only seldom add anything of importance to a graphic.

- Cute little clip arts can easily draw attention away from the data.

Part II
Installation and Configuration

by Till Tantau

This part explains how the system is installed. Typically, someone has already done so for your system, so this part can be skipped; but if this is not the case and you are the poor fellow who has to do the installation, read the present part.



The current candidate for the busy beaver for five states. It is presumed that this Turing machine writes a maximum number of 1's before halting among all Turing machines with five states and the tape alphabet $\{0, 1\}$. Proving this conjecture is an open research problem.

```
\usetikzlibrary {arrows.meta,automata,positioning,shadows}
\begin{tikzpicture}[->,>={Stealth[round]},shorten >=1pt,auto,node distance=2.8cm,on grid,semithick,
                    every state/.style={fill=red,draw=none,circular drop shadow,text=white}]

  \node[initial,state] (A)                    {$q_a$};
  \node[state]         (B) [above right=of A] {$q_b$};
  \node[state]         (D) [below right=of A] {$q_d$};
  \node[state]         (C) [below right=of B] {$q_c$};
  \node[state]         (E) [below=of D]       {$q_e$};

  \path (A) edge              node {0,1,L} (B)
            edge              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) edge [bend left]  node {1,0,R} (A);

   \node [right=1cm,text width=8cm] at (C)
   {
     The current candidate for the busy beaver for five states. It is
     presumed that this Turing machine writes a maximum number of
     $1$'s before halting among all Turing machines with five states
     and the tape alphabet $\{0, 1\}$. Proving this conjecture is an
     open research problem.
   };
\end{tikzpicture}
```

8 Installation

There are different ways of installing PGF, depending on your system and needs, and you may need to install other packages as well, see below. Before installing, you may wish to review the licenses under which the package is distributed, see Section 9.

Typically, the package will already be installed on your system. Naturally, in this case you do not need to worry about the installation process at all and you can skip the rest of this section.

8.1 Package and Driver Versions

This documentation is part of version 3.1.12 of the PGF package. In order to run PGF, you need a reasonably recent TEX installation. When using LATEX, you need the following packages installed (newer versions should also work):

- `xcolor` version 2.00.

With plain TEX, `xcolor` is not needed, but you obviously do not get its (full) functionality.

Currently, PGF requires at least TEX Live 2020 which corresponds to the following backend drivers (earlier versions also work in some cases):

- `luatex` version 1.12.0 or higher.
- `pdftex` version 1.40.21 or higher.
- `dvips` version 2020.1 or higher.

  For inter-picture connections, you need to process pictures using `pdftex` version 1.40.21 or higher running in DVI mode.
- `dvipdfm` version 20200315 or higher.

  For inter-picture connections, you need to process pictures using `pdftex` version 1.40.21 or higher running in DVI mode.
- `dvipdfmx` version 20200315 or higher.
- `dvisvgm` version 2.9.1 or higher.
- `tex4ht` version 2018-07-03 or higher.
- `xetex` version 0.999992 or higher.

Currently, PGF supports the following formats:

- `latex` with complete functionality.
- `plain` with complete functionality, except for graphics inclusion, which works only for pdfTEX.
- `context` with complete functionality, except for graphics inclusion, which works only for pdfTEX.

For more details, see Section 10.

8.2 Installing Prebundled Packages

I do not create or manage prebundled packages of PGF, but, fortunately, nice other people do. I cannot give detailed instructions on how to install these packages, since I do not manage them, but I *can* tell you were to find them. If you have a problem with installing, you might wish to have a look at the Debian page or the MiKTEX page first.

8.2.1 Debian

The command "`apt-get install texlive-pictures`" should do the trick. Sit back and relax.

8.2.2 MiKTeX

For MiKTEX, use the update wizard to install the (latest versions of the) packages called `pgf` and `xcolor`.

8.3 Installation in a texmf Tree

For a permanent installation, you place the files of the PGF package in an appropriate `texmf` tree.

When you ask TEX to use a certain class or package, it usually looks for the necessary files in so-called `texmf` trees. These trees are simply huge directories that contain these files. By default, TEX looks for files in three different `texmf` trees:

- The root `texmf` tree, which is usually located at `/usr/share/texmf/` or `c:\texmf\` or somewhere similar.
- The local `texmf` tree, which is usually located at `/usr/local/share/texmf/` or `c:\localtexmf\` or somewhere similar.
- Your personal `texmf` tree, which is usually located in your home directory at `~/texmf/` or `~/Library/texmf/`.

You should install the packages either in the local tree or in your personal tree, depending on whether you have write access to the local tree. Installation in the root tree can cause problems, since an update of the whole TEX installation will replace this whole tree.

8.3.1 Installation that Keeps Everything Together

Once you have located the right texmf tree, you must decide whether you want to install PGF in such a way that "all its files are kept in one place" or whether you want to be "TDS-compliant", where TDS means "TEX directory structure".

If you want to keep "everything in one place", inside the `texmf` tree that you have chosen create a sub-sub-directory called `texmf/tex/generic/pgf` or `texmf/tex/generic/pgf-3.1.12`, if you prefer. Then place all files of the `pgf` package in this directory. Finally, rebuild TEX's filename database. This is done by running the command `texhash` or `mktexlsr` (they are the same). In MiKTEX, there is a menu option to do this.

8.3.2 Installation that is TDS-Compliant

While the above installation process is the most "natural" one and although I would like to recommend it since it makes updating and managing the PGF package easy, it is not TDS-compliant. If you want to be TDS-compliant, proceed as follows: (If you do not know what TDS-compliant means, you probably do not want to be TDS-compliant.)

The `.tar` file of the `pgf` package contains the following files and directories at its root: `README`, `doc`, `generic`, `plain`, and `latex`. You should "merge" each of the four directories with the following directories `texmf/doc`, `texmf/tex/generic`, `texmf/tex/plain`, and `texmf/tex/latex`. For example, in the `.tar` file the `doc` directory contains just the directory `pgf`, and this directory has to be moved to `texmf/doc/pgf`. The root `README` file can be ignored since it is reproduced in `doc/pgf/README`.

You may also consider keeping everything in one place and using symbolic links to point from the TDS-compliant directories to the central installation.

For a more detailed explanation of the standard installation process of packages, you might wish to consult `http://www.ctan.org/installationadvice/`. However, note that the PGF package does not come with a `.ins` file (simply skip that part).

8.4 Updating the Installation

To update your installation from a previous version, all you need to do is to replace everything in the directory `texmf/tex/generic/pgf` with the files of the new version (or in all the directories where `pgf` was installed, if you chose a TDS-compliant installation). The easiest way to do this is to first delete the old version and then proceed as described above. Sometimes, there are changes in the syntax of certain commands from version to version. If things no longer work that used to work, you may wish to have a look at the release notes and at the change log.

9 Licenses and Copyright

9.1 Which License Applies?

Different parts of the PGF package are distributed under different licenses:

1. The *code* of the package is dual-license. This means that you can decide which license you wish to use when using the PGF package. The two options are:
   (a) You can use the GNU Public License, version 2.
   (b) You can use the LaTeX Project Public License, version 1.3c.
2. The *documentation* of the package is also dual-license. Again, you can choose between two options:
   (a) You can use the GNU Free Documentation License, version 1.2.
   (b) You can use the LaTeX Project Public License, version 1.3c.

The "documentation of the package" refers to all files in the subdirectory `doc` of the `pgf` package. A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-documentation.txt`. All files in other directories are part of the "code of the package". A detailed listing can be found in the file `doc/generic/pgf/licenses/manifest-code.txt`.

In the rest of this section, the licenses are presented. The following text is copyrighted, see the plain text versions of these licenses in the directory `doc/generic/pgf/licenses` for details.

The example picture used in this manual, the Brave GNU World logo, is taken from the Brave GNU World homepage, where it is copyrighted as follows: "Copyright (C) 1999, 2000, 2001, 2002, 2003, 2004 Georg C. F. Greve. Permission is granted to make and distribute verbatim copies of this transcript as long as the copyright and this permission notice appear."

9.2 The GNU Public License, Version 2

9.2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

9.2.2 Terms and Conditions For Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsubsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

9.2.3 No Warranty

10. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

11. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

9.3 The LaTeX Project Public License, Version 1.3c 2006-05-20

9.3.1 Preamble

The LaTeX Project Public License (LPPL) is the primary license under which the LaTeX kernel and the base LaTeX packages are distributed.

You may use this license for any work of which you hold the copyright and which you wish to distribute. This license may be particularly suitable if your work is TeX-related (such as a LaTeX package), but it is written in such a way that you can use it even if your work is unrelated to TeX.

The section 'WHETHER AND HOW TO DISTRIBUTE WORKS UNDER THIS LICENSE', below, gives instructions, examples, and recommendations for authors who are considering distributing their works under this license.

This license gives conditions under which a work may be distributed and modified, as well as conditions under which modified versions of that work may be distributed.

We, the LaTeX3 Project, believe that the conditions below give you the freedom to make and distribute modified versions of your work that conform with whatever technical specifications you wish while maintaining the availability, integrity, and reliability of that work. If you do not see how to achieve your goal while meeting these conditions, then read the document '`cfgguide.tex`' and '`modguide.tex`' in the base LaTeX distribution for suggestions.

9.3.2 Definitions

In this license document the following terms are used:

Work Any work being distributed under this License.

Derived Work Any work that under any applicable law is derived from the Work.

Modification Any procedure that produces a Derived Work under any applicable law – for example, the production of a file containing an original file associated with the Work or a significant portion of such a file, either verbatim or with modifications and/or translated into another language.

Modify To apply any procedure that produces a Derived Work under any applicable law.

Distribution Making copies of the Work available from one person to another, in whole or in part. Distribution includes (but is not limited to) making any electronic components of the Work accessible by file transfer protocols such as FTP or HTTP or by shared file systems such as Sun's Network File System (NFS).

Compiled Work A version of the Work that has been processed into a form where it is directly usable on a computer system. This processing may include using installation facilities provided by the Work, transformations of the Work, copying of components of the Work, or other activities. Note that modification of any installation facilities provided by the Work constitutes modification of the Work.

Current Maintainer A person or persons nominated as such within the Work. If there is no such explicit nomination then it is the 'Copyright Holder' under any applicable law.

Base Interpreter A program or process that is normally needed for running or interpreting a part or the whole of the Work.

A Base Interpreter may depend on external components but these are not considered part of the Base Interpreter provided that each external component clearly identifies itself whenever it is used interactively. Unless explicitly specified when applying the license to the Work, the only applicable Base Interpreter is a 'LaTeX-Format' or in the case of files belonging to the 'LaTeX-format' a program implementing the 'TeX language'.

9.3.3 Conditions on Distribution and Modification

1. Activities other than distribution and/or modification of the Work are not covered by this license; they are outside its scope. In particular, the act of running the Work is not restricted and no requirements are made concerning any offers of support for the Work.

2. You may distribute a complete, unmodified copy of the Work as you received it. Distribution of only part of the Work is considered modification of the Work, and no right to distribute such a Derived Work may be assumed under the terms of this clause.

3. You may distribute a Compiled Work that has been generated from a complete, unmodified copy of the Work as distributed under Clause 2 above, as long as that Compiled Work is distributed in such a way that the recipients may install the Compiled Work on their system exactly as it would have been installed if they generated a Compiled Work directly from the Work.

4. If you are the Current Maintainer of the Work, you may, without restriction, modify the Work, thus creating a Derived Work. You may also distribute the Derived Work without restriction, including Compiled Works generated from the Derived Work. Derived Works distributed in this manner by the Current Maintainer are considered to be updated versions of the Work.

5. If you are not the Current Maintainer of the Work, you may modify your copy of the Work, thus creating a Derived Work based on the Work, and compile this Derived Work, thus creating a Compiled Work based on the Derived Work.

6. If you are not the Current Maintainer of the Work, you may distribute a Derived Work provided the following conditions are met for every component of the Work unless that component clearly states in the copyright notice that it is exempt from that condition. Only the Current Maintainer is allowed to add such statements of exemption to a component of the Work.

(a) If a component of this Derived Work can be a direct replacement for a component of the Work when that component is used with the Base Interpreter, then, wherever this component of the Work identifies itself to the user when used interactively with that Base Interpreter, the replacement component of this Derived Work clearly and unambiguously identifies itself as a modified version of this component to the user when used interactively with that Base Interpreter.

(b) Every component of the Derived Work contains prominent notices detailing the nature of the changes to that component, or a prominent reference to another file that is distributed as part of the Derived Work and that contains a complete and accurate log of the changes.

(c) No information in the Derived Work implies that any persons, including (but not limited to) the authors of the original version of the Work, provide any support, including (but not limited to) the reporting and handling of errors, to recipients of the Derived Work unless those persons have stated explicitly that they do provide such support for the Derived Work.

(d) You distribute at least one of the following with the Derived Work:

i. A complete, unmodified copy of the Work; if your distribution of a modified component is made by offering access to copy the modified component from a designated place, then offering equivalent access to copy the Work from the same or some similar place meets this condition, even though third parties are not compelled to copy the Work along with the modified component;

ii. Information that is sufficient to obtain a complete, unmodified copy of the Work.

7. If you are not the Current Maintainer of the Work, you may distribute a Compiled Work generated from a Derived Work, as long as the Derived Work is distributed to all recipients of the Compiled Work, and as long as the conditions of Clause 6, above, are met with regard to the Derived Work.

8. The conditions above are not intended to prohibit, and hence do not apply to, the modification, by any method, of any component so that it becomes identical to an updated version of that component of the Work as it is distributed by the Current Maintainer under Clause 4, above.

9. Distribution of the Work or any Derived Work in an alternative format, where the Work or that Derived Work (in whole or in part) is then produced by applying some process to that format, does not relax or nullify any sections of this license as they pertain to the results of applying that process.

10. (a) A Derived Work may be distributed under a different license provided that license itself honors the conditions listed in Clause 6 above, in regard to the Work, though it does not have to honor the rest of the conditions in this license.

(b) If a Derived Work is distributed under a different license, that Derived Work must provide sufficient documentation as part of itself to allow each recipient of that Derived Work to honor the restrictions in Clause 6 above, concerning changes from the Work.

11. This license places no restrictions on works that are unrelated to the Work, nor does this license place any restrictions on aggregating such works with the Work by any means.

12. Nothing in this license is intended to, or may be used to, prevent complete compliance by all parties with all applicable laws.

9.3.4 No Warranty

There is no warranty for the Work. Except when otherwise stated in writing, the Copyright Holder provides the Work 'as is', without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the Work is with you. Should the Work prove defective, you assume the cost of all necessary servicing, repair, or correction.

In no event unless required by applicable law or agreed to in writing will The Copyright Holder, or any author named in the components of the Work, or any other party who may distribute and/or modify the Work as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of any use of the Work or out of inability to use the Work (including, but not limited to, loss of data, data being rendered inaccurate, or losses sustained by anyone as a result of any failure of the Work to operate with any other programs), even if the Copyright Holder or said author or said other party has been advised of the possibility of such damages.

9.3.5 Maintenance of The Work

The Work has the status 'author-maintained' if the Copyright Holder explicitly and prominently states near the primary copyright notice in the Work that the Work can only be maintained by the Copyright Holder or simply that it is 'author-maintained'.

The Work has the status 'maintained' if there is a Current Maintainer who has indicated in the Work that they are willing to receive error reports for the Work (for example, by supplying a valid e-mail address). It is not required for the Current Maintainer to acknowledge or act upon these error reports.

The Work changes from status 'maintained' to 'unmaintained' if there is no Current Maintainer, or the person stated to be Current Maintainer of the work cannot be reached through the indicated means of communication for a period of six months, and there are no other significant signs of active maintenance.

You can become the Current Maintainer of the Work by agreement with any existing Current Maintainer to take over this role.

If the Work is unmaintained, you can become the Current Maintainer of the Work through the following steps:

1. Make a reasonable attempt to trace the Current Maintainer (and the Copyright Holder, if the two differ) through the means of an Internet or similar search.
2. If this search is successful, then enquire whether the Work is still maintained.
   (a) If it is being maintained, then ask the Current Maintainer to update their communication data within one month.
   (b) If the search is unsuccessful or no action to resume active maintenance is taken by the Current Maintainer, then announce within the pertinent community your intention to take over maintenance. (If the Work is a LaTeX work, this could be done, for example, by posting to `comp.text.tex`.)
3. (a) If the Current Maintainer is reachable and agrees to pass maintenance of the Work to you, then this takes effect immediately upon announcement.
   (b) If the Current Maintainer is not reachable and the Copyright Holder agrees that maintenance of the Work be passed to you, then this takes effect immediately upon announcement.
4. If you make an 'intention announcement' as described in 2b above and after three months your intention is challenged neither by the Current Maintainer nor by the Copyright Holder nor by other people, then you may arrange for the Work to be changed so as to name you as the (new) Current Maintainer.
5. If the previously unreachable Current Maintainer becomes reachable once more within three months of a change completed under the terms of 3b or 4, then that Current Maintainer must become or remain the Current Maintainer upon request provided they then update their communication data within one month.

A change in the Current Maintainer does not, of itself, alter the fact that the Work is distributed under the LPPL license.

If you become the Current Maintainer of the Work, you should immediately provide, within the Work, a prominent and unambiguous statement of your status as Current Maintainer. You should also announce your new status to the same pertinent community as in 2b above.

9.3.6 Whether and How to Distribute Works under This License

This section contains important instructions, examples, and recommendations for authors who are considering distributing their works under this license. These authors are addressed as 'you' in this section.

9.3.7 Choosing This License or Another License

If for any part of your work you want or need to use *distribution* conditions that differ significantly from those in this license, then do not refer to this license anywhere in your work but, instead, distribute your work under a different license. You may use the text of this license as a model for your own license, but your license should not refer to the LPPL or otherwise give the impression that your work is distributed under the LPPL.

The document '`modguide.tex`' in the base LaTeX distribution explains the motivation behind the conditions of this license. It explains, for example, why distributing LaTeX under the GNU General Public

License (GPL) was considered inappropriate. Even if your work is unrelated to LaTeX, the discussion in '`modguide.tex`' may still be relevant, and authors intending to distribute their works under any license are encouraged to read it.

9.3.8 A Recommendation on Modification Without Distribution

It is wise never to modify a component of the Work, even for your own personal use, without also meeting the above conditions for distributing the modified component. While you might intend that such modifications will never be distributed, often this will happen by accident – you may forget that you have modified that component; or it may not occur to you when allowing others to access the modified version that you are thus distributing it and violating the conditions of this license in ways that could have legal implications and, worse, cause problems for the community. It is therefore usually in your best interest to keep your copy of the Work identical with the public one. Many works provide ways to control the behavior of that work without altering any of its licensed components.

9.3.9 How to Use This License

To use this license, place in each of the components of your work both an explicit copyright notice including your name and the year the work was authored and/or last substantially modified. Include also a statement that the distribution and/or modification of that component is constrained by the conditions in this license.

Here is an example of such a notice and statement:

```
%% pig.dtx
%% Copyright 2005 M. Y. Name
%
% This work may be distributed and/or modified under the
% conditions of the LaTeX Project Public License, either version 1.3
% of this license or (at your option) any later version.
% The latest version of this license is in
%   http://www.latex-project.org/lppl.txt
% and version 1.3 or later is part of all distributions of LaTeX
% version 2005/12/01 or later.
%
% This work has the LPPL maintenance status `maintained'.
%
% The Current Maintainer of this work is M. Y. Name.
%
% This work consists of the files pig.dtx and pig.ins
% and the derived file pig.sty.
```

Given such a notice and statement in a file, the conditions given in this license document would apply, with the 'Work' referring to the three files '`pig.dtx`', '`pig.ins`', and '`pig.sty`' (the last being generated from '`pig.dtx`' using '`pig.ins`'), the 'Base Interpreter' referring to any 'LaTeX-Format', and both 'Copyright Holder' and 'Current Maintainer' referring to the person 'M. Y. Name'.

If you do not want the Maintenance section of LPPL to apply to your Work, change 'maintained' above into 'author-maintained'. However, we recommend that you use 'maintained' as the Maintenance section was added in order to ensure that your Work remains useful to the community even when you can no longer maintain and support it yourself.

9.3.10 Derived Works That Are Not Replacements

Several clauses of the LPPL specify means to provide reliability and stability for the user community. They therefore concern themselves with the case that a Derived Work is intended to be used as a (compatible or incompatible) replacement of the original Work. If this is not the case (e.g., if a few lines of code are reused for a completely different task), then clauses 6b and 6d shall not apply.

9.3.11 Important Recommendations

Defining What Constitutes the Work The LPPL requires that distributions of the Work contain all the files of the Work. It is therefore important that you provide a way for the licensee to determine which

files constitute the Work. This could, for example, be achieved by explicitly listing all the files of the Work near the copyright notice of each file or by using a line such as:

```
% This work consists of all files listed in manifest.txt.
```

in that place. In the absence of an unequivocal list it might be impossible for the licensee to determine what is considered by you to comprise the Work and, in such a case, the licensee would be entitled to make reasonable conjectures as to which files comprise the Work.

9.4 GNU Free Documentation License, Version 1.2, November 2002

9.4.1 Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

9.4.2 Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The **"Document"**, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as **"you"**. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A **"Modified Version"** of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A **"Secondary Section"** is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The **"Invariant Sections"** are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The **"Cover Texts"** are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A **"Transparent"** copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called **"Opaque"**.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The **"Title Page"** means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section **"Entitled XYZ"** means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as **"Acknowledgements"**, **"Dedications"**, **"Endorsements"**, or **"History"**.) To **"Preserve the Title"** of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

9.4.3 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

9.4.4 Copying in Quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

9.4.5 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified

Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any

one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

9.4.6 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

9.4.7 Collection of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

9.4.8 Aggregating with independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

9.4.9 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9.4.10 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will

automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9.4.11 Future Revisions of this License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

9.4.12 Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright ©YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with … Texts". line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

10 Supported Formats

TEX was designed to be a flexible system. This is true both for the *input* for TEX as well as for the *output*. The present section explains which input formats there are and how they are supported by PGF. It also explains which different output formats can be produced.

10.1 Supported Input Formats: LATEX, Plain TEX, ConTEXt

TEX does not prescribe exactly how your input should be formatted. While it is *customary* that, say, an opening brace starts a scope in TEX, this is by no means necessary. Likewise, it is *customary* that environments start with `\begin`, but TEX could not really care less about the exact command name.

Even though TEX can be reconfigured, users can not. For this reason, certain *input formats* specify a set of commands and conventions how input for TEX should be formatted. There are currently three "major" formats: Donald Knuth's original plain TEX format, Leslie Lamport's popular LATEX format, and Hans Hangen's ConTEXt format.

10.1.1 Using the LATEX Format

Using PGF and Ti*k*Z with the LATEX format is easy: You say `\usepackage{pgf}` or `\usepackage{tikz}`. Usually, that is all you need to do, all configuration will be done automatically and (hopefully) correctly.

The style files used for the LATEX format reside in the subdirectory `latex/pgf/` of the PGF-system. Mainly, what these files do is to include files in the directory `generic/pgf`. For example, here is the content of the file `latex/pgf/frontends/tikz.sty`:

```
% Copyright 2019 by Till Tantau
%
% This file may be distributed and/or modified
%
% 1. under the LaTeX Project Public License and/or
% 2. under the GNU Public License.
%
% See the file doc/generic/pgf/licenses/LICENSE for more details.


\RequirePackage{pgf,pgffor}

\input{tikz.code.tex}

\endinput
```

The files in the `generic/pgf` directory do the actual work.

10.1.2 Using the Plain TEX Format

When using the plain TEX format, you say `\input{pgf.tex}` or `\input{tikz.tex}`. Then, instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\pgfpicture` and `\endpgfpicture`.

Unlike for the LATEX format, PGF is not as good at discerning the appropriate configuration for the plain TEX format. In particular, it can only automatically determine the correct output format if you use `pdftex` or `tex` plus `dvips`. For all other output formats you need to set the macro `\pgfsysdriver` to the correct value. See the description of using output formats later on.

Like the LATEX style files, the plain TEX files like `tikz.tex` also just include the correct `tikz.code.tex` file.

10.1.3 Using the ConTEXt Format

When using the ConTEXt format, you say `\usemodule[pgf]` or `\usemodule[tikz]`. As for the plain TEX format you also have to replace the start- and end-of-environment tags as follows: Instead of `\begin{pgfpicture}` and `\end{pgfpicture}` you use `\startpgfpicture` and `\stoppgfpicture`; similarly, instead of `\begin{tikzpicture}` and `\end{tikzpicture}` you use must now use `\starttikzpicture` and `\stoptikzpicture`; and so on for other environments.

The ConTEXt support is very similar to the plain TEX support, so the same restrictions apply: You may have to set the output format directly and graphics inclusion may be a problem.

In addition to `pgf` and `tikz` there also exist modules like `pgfcore` or `pgfmodulematrix`. To use them, you may need to include the module `pgfmod` first (the modules `pgf` and `tikz` both include `pgfmod` for you,

so typically you can skip this). This special module is necessary since old versions of ConTEXt MkII before 2005 satanically restricted the length of module names to 8 characters and PGF's long names are mapped to cryptic 6-letter-names for you by the module `pgfmod`. This restriction was never in place in ConTEXt MkIV and the `pgfmod` module can be safely ignored nowadays.

10.2 Supported Output Formats

An output format is a format in which TEX outputs the text it has typeset. Producing the output is (conceptually) a two-stage process:

1. TEX typesets your text and graphics. The result of this typesetting is mainly a long list of letter–coordinate pairs, plus (possibly) some "special" commands. This long list of pairs is written to something called a `.dvi`-file (informally known as "device-independent file").
2. Some other program reads this `.dvi`-file and translates the letter–coordinate pairs into, say, PostScript commands for placing the given letter at the given coordinate.

The classical example of this process is the combination of `latex` and `dvips`. The `latex` program (which is just the `tex` program called with the LATEX-macros preinstalled) produces a `.dvi`-file as its output. The `dvips` program takes this output and produces a `.ps`-file (a PostScript file). Possibly, this file is further converted using, say, `ps2pdf`, whose name is supposed to mean "PostScript to PDF". Another example of programs using this process is the combination of `tex` and `dvipdfm`. The `dvipdfm` program takes a `.dvi`-file as input and translates the letter–coordinate pairs therein into PDF-commands, resulting in a `.pdf` file directly. Finally, the `tex4ht` is also a program that takes a `.dvi`-file and produces an output, this time it is a `.html` file. The programs `pdftex` and `pdflatex` are special: They directly produce a `.pdf`-file without the intermediate `.dvi`-stage. However, from the programmer's point of view they behave exactly as if there was an intermediate stage.

Normally, TEX only produces letter–coordinate pairs as its "output". This obviously makes it difficult to draw, say, a curve. For this, "special" commands can be used. Unfortunately, these special commands are not the same for the different programs that process the `.dvi`-file. Indeed, every program that takes a `.dvi`-file as input has a totally different syntax for the special commands.

One of the main jobs of PGF is to "abstract away" the difference in the syntax of the different programs. However, this means that support for each program has to be "programmed", which is a time-consuming and complicated process.

10.2.1 Selecting the Backend Driver

When TEX typesets your document, it does not know which program you are going to use to transform the `.dvi`-file. If your `.dvi`-file does not contain any special commands, this would be fine; but these days almost all `.dvi`-files contain lots of special commands. It is thus necessary to tell TEX which program you are going to use later on.

Unfortunately, there is no "standard" way of telling this to TEX. For the LATEX format a sophisticated mechanism exists inside the `graphics` package and PGF plugs into this mechanism. For other formats and when this plugging does not work as expected, it is necessary to tell PGF directly which program you are going to use. This is done by redefining the macro `\pgfsysdriver` to an appropriate value *before* you load `pgf`. If you are going to use the `dvips` program, you set this macro to the value `pgfsys-dvips.def`; if you use `pdftex` or `pdflatex`, you set it to `pgfsys-pdftex.def`; and so on. In the following, details of the support of the different programs are discussed.

10.2.2 Producing PDF Output

PGF supports multiple programs that produce PDF output (PDF means "portable document format" and was invented by the Adobe company): `dvipdfm`, `pdftex`, `luatex`, and `xetex` (via `dvipdfmx`). The `pdflatex` program is the same as the `pdftex` program: it uses a different input format, but the output is exactly the same.

File `pgfsys-pdftex.def`

This is the driver file for use with pdfTEX, that is, with the `pdftex` or `pdflatex` command. It includes `pgfsys-common-pdf.def`.

This driver has a lot of functionality. (Almost) everything PGF "can do at all" is implemented in this driver.

File pgfsys-dvipdfm.def

This is a driver file for use with (la)tex followed by dvipdfm. It includes pgfsys-common-pdf.def.

This driver supports most of PGF's features, but there are some restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion and does not support masking.
2. In plain TeX mode it does not support image inclusion.

File pgfsys-xetex.def

This is a driver file for use with xe(la)tex followed by xdvipdfmx. This driver supports largely the same operations as the dvipdfm driver.

It is also possible to produce a .pdf-file by first producing a PostScript file (see below) and then using a PostScript-to-PDF conversion program like ps2pdf or Acrobat Distiller.

10.2.3 Producing PostScript Output

File pgfsys-dvips.def

This is a driver file for use with (la)tex followed by dvips. It includes pgfsys-common-postscript.def.

This driver also supports most of PGF's features, except for the following restrictions:

1. In LaTeX mode it uses graphicx for the graphics inclusion. Image masking is supported if the PostScript output is further processed with ps2pdf to produce PDF.
2. In plain TeX mode it does not support image inclusion.
3. Functional shadings are approximated with Type-0 functions (sampled functions), because Type-4 functions are not available in the latest (version 3) PostScript language definition. Due to their fixed resolution, Type-0 functional shadings are of lesser quality at higher zoom levels as compared to functional shadings from PDF producing drivers. Axial and radial shadings are fully supported. The same output quality (smooth shadings) is achieved as with drivers that produce PDF output.
4. Although fully supported, opacity and fadings are PDF features that become visible only after further processing the PostScript output with ps2pdf. Note that newer Ghostscript versions are necessary for producing opacity in the PDF output. Also, beginning with version 9.52 of Ghostscript, command line option -dALLOWPSTRANSPARENCY must be added:

   ```
   ps2pdf -dALLOWPSTRANSPARENCY example.ps
   ```

5. For remembering of pictures (inter-picture connections) you need to use a recent version of pdftex running in DVI-mode.

10.2.4 Producing SVG Output

File pgfsys-dvisvgm.def

This driver converts DVI files to SVG file, including text and fonts. When you select this driver, PGF will output the required raw SVG code for the pictures it produces.

Since the graphics package does not (yet?) support this driver directly, there is special rule for this driver in LaTeX: If the option dvisvgm is given to the tikz package, this driver gets selected (normally, the driver selected by graphics would be used). For packages like beamer that load PGF themselves, this means that the option dvisvgm should be given to the document class.

```
% example.tex
\documentclass[dvisvgm]{minimal}

\usepackage{tikz}

\begin{document}
Hello \tikz [baseline] \fill [fill=blue!80!black] (0,.75ex) circle[radius=.75ex];
\end{document}
```

And then run

```
latex example
dvisvgm example
```

or better

```
lualatex --output-format=dvi example
dvisvgm example
```

(This is "better" since it gives you access to the full power of LuaTEX inside your TEX-file. In particular, Ti*k*Z is able to run graph drawing algorithms in this case.)

Unlike the `tex4ht` driver below, this driver has full support of text nodes.

File `pgfsys-tex4ht.def`

This is a driver file for use with the `tex4ht` program. It is selected automatically when the `tex4ht` style or command is loaded. It includes `pgfsys-common-svg.def`.

The `tex4ht` program converts `.dvi`-files to `.html`-files. While the HTML-format cannot be used to draw graphics, the SVG-format can. This driver will ask PGF to produce an SVG-picture for each PGF graphic in your text.

When using this driver you should be aware of the following restrictions:

1. In LaTeX mode it uses `graphicx` for the graphics inclusion.
2. In plain TeX mode it does not support image inclusion.
3. Remembering of pictures (inter-picture connections) is not supported.
4. Text inside `pgfpicture`s is not supported very well. The reason is that the SVG specification currently does not support text very well and, although it is possible to "escape back" to HTML, Ti*k*Z has then to guess what size the text rendered by the browser would have.
5. Unlike for other output formats, the bounding box of a picture "really crops" the picture.
6. Matrices do not work.
7. Functional shadings are not supported.

The driver basically works as follows: When a `{pgfpicture}` is started, appropriate `\special` commands are used to directed the output of `tex4ht` to a new file called `\jobname-xxx.svg`, where `xxx` is a number that is increased for each graphic. Then, till the end of the picture, each (system layer) graphic command creates a special that inserts appropriate SVG literal text into the output file. The exact details are a bit complicated since the imaging model and the processing model of PostScript/PDF and SVG are not quite the same; but they are "close enough" for PGF's purposes.

Because text is not supported very well in the SVG standard, you may wish to use the following options to modify the way text is handled:

`/pgf/tex4ht node/escape=`⟨*boolean*⟩ (default `false`)

Selects the rendering method for a text node with the tex4ht driver.

When this key is set to `false`, text is translated into SVG text, which is somewhat limited: simple characters (letters, numerals, punctuation, $\sum$, $\int$, ...), subscripts and superscripts (but not subsubscripts) will display but everything else will be filtered out, ignored or will produce invalid HTML code (in the worst case). This means that two kind of texts render reasonably well:

1. First, plain text without math mode, special characters or anything else special.
2. Second, *very* simple mathematical text that contains subscripts or superscripts. Even then, variables are not correctly set in italics and, in general, text simple does not look very nice.

If you use text that contains anything special, even something as simple as `$\alpha$`, this may corrupt the graphic.

```
\tikz \node[draw,/pgf/tex4ht node/escape=false] {Example : $(a+b)^2=a^2+2ab+b^2$};
```

When you write `node[/pgf/tex4ht node/escape=true] {`⟨*text*⟩`}`, PGF escapes back to HTML to render the ⟨*text*⟩. This method produces valid HTML code in most cases and the support for complicated text nodes is much better since code that renders well outside a `{pgfpicture}`, should also render well inside a text node. Another advantage is that inside text nodes with fixed width, HTML will produce line breaks for long lines. On the other hand, you need a browser with good SVG support to display the picture. Also, the text will display differently, depending on your browsers,

the fonts you have on your system and your settings. Finally, PGF has to guess the size of the text rendered by the browser to scale it and prevent it from sticking from the node. When it fails, the text will be either cropped or too small.

```
\tikz \node[draw,/pgf/tex4ht node/escape=true]
  {Example : $\int_0^\infty\frac{1}{1+t^2}dt=\frac{\pi}{2}$};
```

/pgf/tex4ht node/css=⟨*filename*⟩ (default `\jobname`)

This option allows you to tell the browser what CSS file it should use to style the display of the node (only with `tex4ht node/escape=true`).

/pgf/tex4ht node/class=⟨*class name*⟩ (default `foreignobject`)

This option allows you to give a class name to the node, allowing it to be styled by a CSS file (only with `tex4ht node/escape=true`).

/pgf/tex4ht node/id=⟨*id name*⟩ (default `\jobname picture number-node number`)

This option allows you to give a unique id to the node, allowing it to be styled by a CSS file (only with `tex4ht node/escape=true`).

10.2.5 Producing Perfectly Portable DVI Output

File pgfsys-dvi.def

This is a driver file that can be used with any output driver, except for `tex4ht`.

The driver will produce perfectly portable `.dvi` files by composing all pictures entirely of black rectangles, the basic and only graphic shape supported by the TEX core. Even straight, but slanted lines are tricky to get right in this model (they need to be composed of lots of little squares).

Naturally, *very little* is possible with this driver. In fact, so little is possible that it is easier to list what is possible:

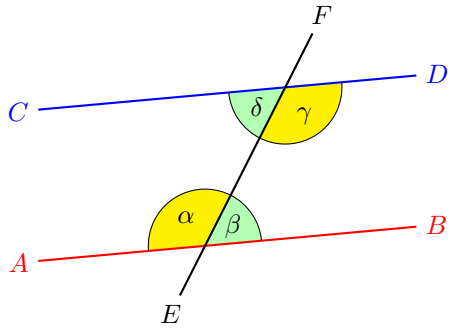
- Text boxes can be placed in the normal way.
- Lines and curves can be drawn (stroked). If they are not horizontal or vertical, they are composed of hundreds of small rectangles.
- Lines of different width are supported.
- Transformations are supported.

Note that, say, even filling is not supported! (Let alone color or anything fancy.)

This driver has only one real application: It might be useful when you only need horizontal or vertical lines in a picture. Then, the results are quite satisfactory.

Part III
TikZ ist *kein* Zeichenprogramm

by Till Tantau



When we assume that AB and CD are parallel, i. e., $AB \parallel CD$, then $\alpha = \gamma$ and $\beta = \delta$.

```
\usetikzlibrary {angles,calc,quotes}
\begin{tikzpicture}[angle radius=.75cm]

  \node (A) at (-2,0)      [red,left]   {$A$};
  \node (B) at ( 3,.5)     [red,right]  {$B$};
  \node (C) at (-2,2)      [blue,left]  {$C$};
  \node (D) at ( 3,2.5)    [blue,right] {$D$};
  \node (E) at (60:-5mm)   [below]      {$E$};
  \node (F) at (60:3.5cm)  [above]      {$F$};

  \coordinate (X) at (intersection cs:first line={(A)--(B)}, second line={(E)--(F)});
  \coordinate (Y) at (intersection cs:first line={(C)--(D)}, second line={(E)--(F)});

  \path
    (A) edge [red, thick]  (B)
    (C) edge [blue, thick] (D)
    (E) edge [thick]       (F)
      pic ["$\alpha$", draw, fill=yellow]   {angle = F--X--A}
      pic ["$\beta$",  draw, fill=green!30] {angle = B--X--F}
      pic ["$\gamma$", draw, fill=yellow]   {angle = E--Y--D}
      pic ["$\delta$", draw, fill=green!30] {angle = C--Y--E};

  \node at ($ (D)!.5!(B) $) [right=1cm,text width=6cm,rounded corners,fill=red!20,inner sep=1ex]
    {
      When we assume that $\color{red}AB$ and $\color{blue}CD$ are
      parallel, i.\,e., ${\color{red}AB} \mathbin{\|} \color{blue}CD$,
      then $\alpha = \gamma$ and $\beta = \delta$.
    };
\end{tikzpicture}
```

11 Design Principles

This section describes the design principles behind the Ti*k*Z frontend, where Ti*k*Z means "Ti*k*Z ist *kein* Zeichenprogramm". To use Ti*k*Z, as a LaTeX user say `\usepackage{tikz}` somewhere in the preamble, as a plain TeX user say `\input{tikz.tex}`. Ti*k*Z's job is to make your life easier by providing an easy-to-learn and easy-to-use syntax for describing graphics.

The commands and syntax of Ti*k*Z were influenced by several sources. The basic command names and the notion of path operations is taken from METAFONT, the option mechanism comes from PSTRICKS, the notion of styles is reminiscent of SVG, the graph syntax is taken from GRAPHVIZ. To make it all work together, some compromises were necessary. I also added some ideas of my own, like coordinate transformations.

The following basic design principles underlie Ti*k*Z:

1. Special syntax for specifying points.
2. Special syntax for path specifications.
3. Actions on paths.
4. Key–value syntax for graphic parameters.
5. Special syntax for nodes.
6. Special syntax for trees.
7. Special syntax for graphs.
8. Grouping of graphic parameters.
9. Coordinate transformation system.

11.1 Special Syntax For Specifying Points

Ti*k*Z provides a special syntax for specifying points and coordinates. In the simplest case, you provide two TeX dimensions, separated by commas, in round brackets as in `(1cm,2pt)`.

You can also specify a point in polar coordinates by using a colon instead of a comma as in `(30:1cm)`, which means "1cm in a 30 degrees direction".

If you do not provide a unit, as in `(2,1)`, you specify a point in PGF's xy-coordinate system. By default, the unit x-vector goes 1cm to the right and the unit y-vector goes 1cm upward.

By specifying three numbers as in `(1,1,1)` you specify a point in PGF's xyz-coordinate system.

It is also possible to use an anchor of a previously defined shape as in `(first node.south)`.

You can add two plus signs before a coordinate as in `++(1cm,0pt)`. This means "1cm to the right of the last point used". This allows you to easily specify relative movements. For example, `(1,0) ++(1,0) ++(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(2,1)`.

Finally, instead of two plus signs, you can also add a single one. This also specifies a point in a relative manner, but it does not "change" the current point used in subsequent relative commands. For example, `(1,0) +(1,0) +(0,1)` specifies the three coordinates `(1,0)`, then `(2,0)`, and `(1,1)`.

11.2 Special Syntax For Path Specifications

When creating a picture using Ti*k*Z, your main job is the specification of *paths*. A path is a series of straight or curved lines, which need not be connected. Ti*k*Z makes it easy to specify paths, partly using the syntax of METAPOST. For example, to specify a triangular path you use

```
(5pt,0pt) -- (0pt,0pt) -- (0pt,5pt) -- cycle
```

and you get ◺ when you draw this path.

11.3 Actions on Paths

A path is just a series of straight and curved lines, but it is not yet specified what should happen with it. One can *draw* a path, *fill* a path, *shade* it, *clip* it, or do any combination of these. Drawing (also known as *stroking*) can be thought of as taking a pen of a certain thickness and moving it along the path, thereby drawing on the canvas. Filling means that the interior of the path is filled with a uniform color. Obviously, filling makes sense only for *closed* paths and a path is automatically closed prior to filling, if necessary.

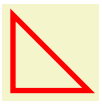
Given a path as in `\path (0,0) rectangle (2ex,1ex);`, you can draw it by adding the `draw` option as in `\path[draw] (0,0) rectangle (2ex,1ex);`, which yields ▭. The `\draw` command is just an abbreviation for `\path[draw]`. To fill a path, use the `fill` option or the `\fill` command, which is an abbreviation for `\path[fill]`. The `\filldraw` command is an abbreviation for `\path[fill,draw]`. Shading is caused by the `shade` option (there are `\shade` and `\shadedraw` abbreviations) and clipping by the `clip` option. There is also a `\clip` command, which does the same as `\path[clip]`, but not commands like `\drawclip`. Use, say, `\draw[clip]` or `\path[draw,clip]` instead.

All of these commands can only be used inside `{tikzpicture}` environments.

Ti*k*Z allows you to use different colors for filling and stroking.

11.4 Key–Value Syntax for Graphic Parameters

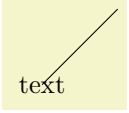
Whenever Ti*k*Z draws or fills a path, a large number of graphic parameters influences the rendering. Examples include the colors used, the dashing pattern, the clipping area, the line width, and many others. In Ti*k*Z, all these options are specified as lists of so called key–value pairs, as in `color=red`, that are passed as optional parameters to the path drawing and filling commands. This usage is similar to PSTRICKS. For example, the following will draw a thick, red triangle;



```
\tikz \draw[line width=2pt,color=red] (1,0) -- (0,0) -- (0,1) -- cycle;
```

11.5 Special Syntax for Specifying Nodes

Ti*k*Z introduces a special syntax for adding text or, more generally, nodes to a graphic. When you specify a path, add nodes as in the following example:




```
\tikz \draw (1,1) node {text} -- (2,2);
```

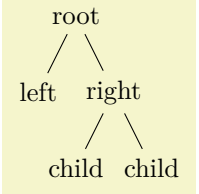
Nodes are inserted at the current position of the path, but either *after* (the default) or *before* the complete path is rendered. When special options are given, as in `\draw (1,1) node[circle,draw] {text};`, the text is not just put at the current position. Rather, it is surrounded by a circle and this circle is "drawn".

You can add a name to a node for later reference either by using the option `name=`⟨*node name*⟩ or by stating the node name in parentheses outside the text as in `node[circle](name){text}`.

Predefined shapes include `rectangle`, `circle`, and `coordinate`, but it is possible (though a bit challenging) to define new shapes.

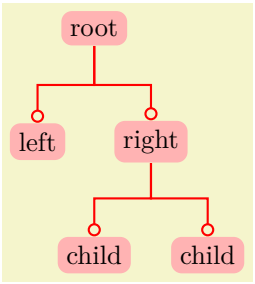
11.6 Special Syntax for Specifying Trees

The "node syntax" can also be used to draw trees: A `node` can be followed by any number of children, each introduced by the keyword `child`. The children are nodes themselves, each of which may have children in turn.

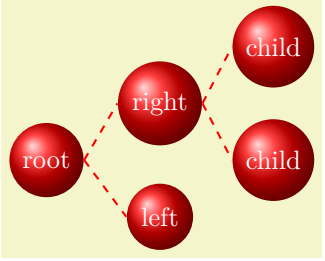



```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Since trees are made up from nodes, it is possible to use options to modify the way trees are drawn. Here are two examples of the above tree, redrawn with different options:



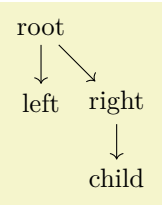

```
\usetikzlibrary {arrows.meta,trees}
\begin{tikzpicture}
  [edge from parent fork down, sibling distance=15mm, level distance=15mm,
   every node/.style={fill=red!30,rounded corners},
   edge from parent/.style={red,-{Circle[open]},thick,draw}]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [parent anchor=east,child anchor=west,grow=east,
   sibling distance=15mm, level distance=15mm,
   every node/.style={ball color=red,circle,text=white},
   edge from parent/.style={draw,dashed,thick,red}]
  \node {root}
      child {node {left}}
      child {node {right}
        child {node {child}}
        child {node {child}}
      };
\end{tikzpicture}
```

11.7 Special Syntax for Graphs

The `\node` command gives you fine control over where nodes should be placed, what text they should use, and what they should look like. However, when you draw a graph, you typically need to create numerous fairly similar nodes that only differ with respect to the name they show. In these cases, the `graph` syntax can be used, which is another syntax layer build "on top" of the node syntax.



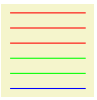

```
\usetikzlibrary {graphs}
\tikz \graph [grow down, branch right] {
  root -> { left, right -> {child, child} }
};
```

The syntax of the `graph` command extends the so-called DOT-notation used in the popular GRAPHVIZ program.

Depending on the version of TEX you use (it must allow you to call Lua code, which is the case for LuaTEX), you can also ask Ti*k*Z to do automatically compute good positions for the nodes of a graph using one of several integrated *graph drawing algorithms*.

11.8 Grouping of Graphic Parameters

Graphic parameters should often apply to several path drawing or filling commands. For example, we may wish to draw numerous lines all with the same line width of 1pt. For this, we put these commands in a `{scope}` environment that takes the desired graphic options as an optional parameter. Naturally, the specified graphic parameters apply only to the drawing and filling commands inside the environment. Furthermore, nested `{scope}` environments or individual drawing commands can override the graphic parameters of outer `{scope}` environments. In the following example, three red lines, two green lines, and one blue line are drawn:



```
\begin{tikzpicture}
  \begin{scope}[color=red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm, 8mm) -- (10mm, 8mm);
    \draw (0mm, 6mm) -- (10mm, 6mm);
  \end{scope}
  \begin{scope}[color=green]
    \draw              (0mm, 4mm) -- (10mm, 4mm);
    \draw              (0mm, 2mm) -- (10mm, 2mm);
    \draw[color=blue] (0mm, 0mm) -- (10mm, 0mm);
  \end{scope}
\end{tikzpicture}
```

The `{tikzpicture}` environment itself also behaves like a `{scope}` environment, that is, you can specify graphic parameters using an optional argument. These optional apply to all commands in the picture.

11.9 Coordinate Transformation System

Ti*k*Z supports both PGF's *coordinate* transformation system to perform transformations as well as *canvas* transformations, a more low-level transformation system. (For details on the difference between coordinate transformations and canvas transformations see Section 99.4.)

The syntax is set up in such a way that it is harder to use canvas transformations than coordinate transformations. There are two reasons for this: First, the canvas transformation must be used with great care and often results in "bad" graphics with changing line width and text in wrong sizes. Second, PGF loses track of where nodes and shapes are positioned when canvas transformations are used. So, in almost all circumstances, you should use coordinate transformations rather than canvas transformations.

12 Hierarchical Structures: Package, Environments, Scopes, and Styles

The present section explains how your files should be structured when you use Ti*k*Z. On the top level, you need to include the `tikz` package. In the main text, each graphic needs to be put in a `{tikzpicture}` environment. Inside these environments, you can use `{scope}` environments to create internal groups. Inside the scopes you use `\path` commands to actually draw something. On all levels (except for the package level), graphic options can be given that apply to everything within the environment.

12.1 Loading the Package and the Libraries

```
\usepackage{tikz} % LaTeX
\input{tikz.tex} % plain TeX
\usemodule[tikz]  % ConTeXt
```

This package does not have any options.

This will automatically load the PGF and the `pgffor` package.

PGF needs to know what TEX driver you are intending to use. In most cases PGF is clever enough to determine the correct driver for you; this is true in particular if you use LATEX. One situation where PGF cannot know the driver "by itself" is when you use plain TEX or ConTEXt together with `dvipdfm`. In this case, you have to write `\def\pgfsysdriver{pgfsys-dvipdfm.def}` *before* you input `tikz.tex`.

`\usetikzlibrary{`⟨*list of libraries*⟩`}`

Once Ti*k*Z has been loaded, you can use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets, which is something ConTEXt users will like. If you try to load a library a second time, nothing will happen.

Example: `\usetikzlibrary{arrows.meta}`

The above command will load a whole bunch of extra arrow tip definitions.

What this command does is to load the file `tikzlibrary`⟨*library*⟩`.code.tex` for each ⟨*library*⟩ in the ⟨*list of libraries*⟩. If this file does not exist, the file `pgflibrary`⟨*library*⟩`.code.tex` is loaded instead. If this file also does not exist, an error message is printed. Thus, to write your own library file, all you need to do is to place a file of the appropriate name somewhere where TEX can find it. LATEX, plain TEX, and ConTEXt users can then use your library.

12.2 Creating a Picture

12.2.1 Creating a Picture Using an Environment

The "outermost" scope of Ti*k*Z is the `{tikzpicture}` environment. You may give drawing commands only inside this environment, giving them outside (as is possible in many other packages) will result in chaos.

In Ti*k*Z, the way graphics are rendered is strongly influenced by graphic options. For example, there is an option for setting the color used for drawing, another for setting the color used for filling, and also more obscure ones like the option for setting the prefix used in the filenames of temporary files written while plotting functions using an external program. The graphic options are specified in *key lists*, see Section 12.4 below for details. All graphic options are local to the `{tikzpicture}` to which they apply.

```
\begin{tikzpicture}⟨animations spec⟩[⟨options⟩]
  ⟨environment contents⟩
\end{tikzpicture}
```

All Ti*k*Z commands should be given inside this environment, except for the `\tikzset` command. You cannot use graphics commands like the low-level command `\pgfpathmoveto` outside this environment and doing so will result in chaos. For Ti*k*Z, commands like `\path` are only defined inside this environment, so there is little chance that you will do something wrong here.

When this environment is encountered, the ⟨*options*⟩ are parsed, see Section 12.4. All options given here will apply to the whole picture. Before the options you can specify animation commands, provided that the `animations` library is loaded, see Section 26 for details.

Next, the contents of the environment is processed and the graphic commands therein are put into a box. Non-graphic text is suppressed as well as possible, but non-PGF commands inside a `{tikzpicture}` environment should not produce any "output" since this may totally scramble the positioning system of the backend drivers. The suppressing of normal text, by the way, is done by temporarily switching the font to `\nullfont`. You can, however, "escape back" to normal TEX typesetting. This happens, for example, when you specify a node.

At the end of the environment, PGF tries to make a good guess at the size of a bounding box of the graphic and then resizes the picture box such that the box has this size. To "make its guess", every time PGF encounters a coordinate, it updates the bounding box's size such that it encompasses all these coordinates. This will usually give a good approximation of the bounding box, but will not always be accurate. First, the line thickness of diagonal lines is not taken into account correctly. Second, control points of a curve often lie far "outside" the curve and make the bounding box too large. In this case, you should use the `[use as bounding box]` option.

The following key influences the baseline of the resulting picture:

`/tikz/baseline=`⟨*dimension or coordinate or* ***default***⟩ (default `0pt`)

Normally, the lower end of the picture is put on the baseline of the surrounding text. For example, when you give the code `\tikz\draw(0,0)circle[radius=.5ex];`, PGF will find out that the lower end of the picture is at $-.5\text{ex} - 0.2\text{pt}$ (the 0.2pt are half the line width, which is 0.4pt) and that the upper end is at $.5\text{ex} + .2\text{pt}$. Then, the lower end will be put on the baseline, resulting in the following: ○.

Using this option, you can specify that the picture should be raised or lowered such that the height ⟨*dimension*⟩ is on the baseline. For example, `\tikz[baseline=0pt]\draw(0,0)circle[radius=.5ex];` yields ○ since, now, the baseline is on the height of the x-axis.

This options is often useful for "inlined" graphics as in

$A \twoheadrightarrow B$

```
$A \mathbin{\tikz[baseline] \draw[->>] (0pt,.5ex) -- (3ex,.5ex);} B$
```

Instead of a ⟨*dimension*⟩ you can also provide a coordinate in parentheses. Then the effect is to put the baseline on the y-coordinate that the given ⟨*coordinate*⟩ has *at the end of the picture*. This means that, at the end of the picture, the ⟨*coordinate*⟩ is evaluated and then the baseline is set to the y-coordinate of the resulting point. This makes it easy to reference the y-coordinate of, say, the baseline of nodes.

Hello ~~world.~~

```
\usetikzlibrary {shapes.misc}
Hello
\tikz[baseline=(X.base)]
  \node [cross out,draw] (X) {world.};
```

Top align:

```
Top align:
\tikz[baseline=(current bounding box.north)]
  \draw (0,0) rectangle (1cm,1ex);
```

Use `baseline=default` to reset the `baseline` option to its initial configuration.

`/tikz/execute at begin picture=`⟨*code*⟩ (no default)

This option causes ⟨*code*⟩ to be executed at the beginning of the picture. This option must be given in the argument of the `{tikzpicture}` environment itself since this option will not have an effect otherwise. After all, the picture has already "started" later on. The effect of multiply setting this option accumulates.

This option is mainly used in styles like the `every picture` style to execute certain code at the start of a picture.

`/tikz/execute at end picture=`⟨*code*⟩ (no default)

This option installs ⟨*code*⟩ that will be executed at the end of the picture. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{tikzpicture}` environment.




```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[execute at end picture=%
  {
    \begin{pgfonlayer}{background}
      \path[fill=yellow,rounded corners]
        (current bounding box.south west) rectangle
        (current bounding box.north east);
    \end{pgfonlayer}
  }]
  \node at (0,0) {X};
  \node at (2,1) {Y};
\end{tikzpicture}
```

All options "end" at the end of the picture. To set an option "globally" change the following style:

`/tikz/every picture` (style, initially empty)

This style is installed at the beginning of each picture.

```
\tikzset{every picture/.style=semithick}
```

Note that you should not use `\tikzset` to set options directly. For instance, if you want to use a line width of `1pt` by default, do not try to say `\tikzset{line width=1pt}` at the beginning of your document. This will not work since the line width is changed in many places. Instead, say

```
\tikzset{every picture/.style={line width=1pt}}
```

This will have the desired effect.

`\tikzpicture[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\endtikzpicture`

The plain TEX version of the environment.

`\starttikzpicture[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\stoptikzpicture`

The ConTEXt version of the environment.

12.2.2 Creating a Picture Using a Command

The following command is an alternative to `{tikzpicture}` that is particular useful for graphics consisting of a single or few commands.

`\tikz`⟨*animations spec*⟩`[`⟨*options*⟩`]{`⟨*path commands*⟩`}`

This command places the ⟨*path commands*⟩ inside a `{tikzpicture}` environment. The ⟨*path commands*⟩ may contain paragraphs and fragile material (like verbatim text).

If there is only one path command, it need not be surrounded by curly braces, if there are several, you need to add them (this is similar to the `\foreach` statement and also to the rules in programming languages like Java or C concerning the placement of curly braces).

Example: `\tikz{\draw (0,0) rectangle (2ex,1ex);}` yields ▭

Example: `\tikz \draw (0,0) rectangle (2ex,1ex);` yields ▭

12.2.3 Handling Catcodes and the Babel Package

Inside a Ti*k*Z picture, most symbols need to have the category code 12 (normal text) in order to ensure that the parser works properly. This is typically not the case when packages like `babel` are used, which change catcodes aggressively.

To solve this problem, Ti*k*Z provides a small library also called `babel` (which can, however, also be used together with any other package that globally changes category codes). What it does is to reset the category codes at the beginning of every `{tikzpicture}` and to restore them at the beginning of every node. In almost all cases, this is exactly what you would expect and need, so I recommend to always load this library by saying `\usetikzlibrary{babel}`. For details on what, exactly, happens with the category codes, see Section 44.

12.2.4 Adding a Background

By default, pictures do not have any background, that is, they are "transparent" on all parts on which you do not draw anything. You may instead wish to have a colored background behind your picture or a black frame around it or lines above and below it or some other kind of decoration.

Since backgrounds are often not needed at all, the definition of styles for adding backgrounds has been put in the library package `backgrounds`. This package is documented in Section 45.

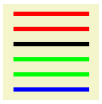
12.3 Using Scopes to Structure a Picture

Inside a `{tikzpicture}` environment you can create scopes using the `{scope}` environment. This environment is available only inside the `{tikzpicture}` environment, so once more, there is little chance of doing anything wrong.

12.3.1 The Scope Environment

`\begin{scope}`⟨*animations spec*⟩`[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{scope}`

All ⟨*options*⟩ are local to the ⟨*environment contents*⟩. Furthermore, the clipping path is also local to the environment, that is, any clipping done inside the environment "ends" at its end.



```
\begin{tikzpicture}[ultra thick]
  \begin{scope}[red]
    \draw (0mm,10mm) -- (10mm,10mm);
    \draw (0mm,8mm) -- (10mm,8mm);
  \end{scope}
  \draw (0mm,6mm) -- (10mm,6mm);
  \begin{scope}[green]
    \draw (0mm,4mm) -- (10mm,4mm);
    \draw (0mm,2mm) -- (10mm,2mm);
    \draw[blue] (0mm,0mm) -- (10mm,0mm);
  \end{scope}
\end{tikzpicture}
```

`/tikz/name=`⟨*scope name*⟩ (no default)

Assigns a name to a scope reference in animations. The name is a "high-level" name that drivers do not see, so you can use spaces, number, letters, in a name, but you should *not* use any punctuation like a dot, a comma, or a colon.

The following style influences scopes:

`/tikz/every scope` (style, initially empty)

This style is installed at the beginning of every scope.

The following options are useful for scopes:

`/tikz/execute at begin scope=`⟨*code*⟩ (no default)

This option install some code that will be executed at the beginning of the scope. This option must be given in the argument of the `{scope}` environment.

The effect applies only to the current scope, not to subscopes.

`/tikz/execute at end scope=`⟨*code*⟩ (no default)

This option installs some code that will be executed at the end of the current scope. Using this option multiple times will cause the code to accumulate. This option must also be given in the optional argument of the `{scope}` environment.

Again, the effect applies only to the current scope, not to subscopes.

`\scope`⟨*animations spec*⟩`[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\endscope`

The plain TEX version of the environment.

\startscope⟨*animations spec*⟩[⟨*options*⟩]
⟨*environment contents*⟩
\stopscope

The ConTEXt version of the environment.

12.3.2 Shorthand for Scope Environments

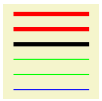
There is a small library that makes using scopes a bit easier:

TikZ Library scopes

```
\usetikzlibrary{scopes} % LaTeX and plain TeX
\usetikzlibrary[scopes] % ConTeXt
```

This library defines a shorthand for starting and ending {scope} environments.

When this library is loaded, the following happens: At certain places inside a TikZ picture, it is allowed to start a scope just using a single brace, provided the single brace is followed by options in square brackets:



```
\usetikzlibrary {scopes}
\begin{tikzpicture}
  { [ultra thick]
    { [red]
      \draw (0mm,10mm) -- (10mm,10mm);
      \draw (0mm,8mm)  -- (10mm,8mm);
    }
    \draw (0mm,6mm) -- (10mm,6mm);
  }
  { [green]
    \draw (0mm,4mm) -- (10mm,4mm);
    \draw (0mm,2mm) -- (10mm,2mm);
    \draw[blue] (0mm,0mm) -- (10mm,0mm);
  }
\end{tikzpicture}
```

In the above example, { [ultra thick] actually causes a \begin{scope}[ultra thick] to be inserted, and the corresponding closing } causes an \end{scope} to be inserted.

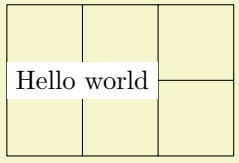
The "certain places" where an opening brace has this special meaning are the following: First, right after the semicolon that ends a path. Second, right after the end of a scope. Third, right at the beginning of a scope, which includes the beginning of a picture. Also note that some square bracket must follow, otherwise the brace is treated as a normal TEX scope.

12.3.3 Single Command Scopes

In some situations it is useful to create a scope for a single command. For instance, when you wish to use algorithm graph drawing in order to layout a tree, the path of the tree needs to be surrounded by a scope whose only purpose is to take a key that selects a layout for the scope. Similarly, in order to put something on a background layer, a scope needs to be created. In such cases, where it will cumbersome to create a \begin{scope} and \end{scope} pair just for a single command, the \scoped command may be useful:

\scoped⟨*animations spec*⟩[⟨*options*⟩]⟨*path command*⟩

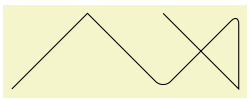
This command works like \tikz, only you can use it inside a {tikzpicture}. It will take the following ⟨*path command*⟩ and put it inside a {scope} with the ⟨*options*⟩ set. The ⟨*path command*⟩ may either be a single command ended by a semicolon or it may contain multiple commands, but then they must be surrounded by curly braces.




```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  \node [fill=white] at (1,1) {Hello world};
  \scoped [on background layer]
    \draw (0,0) grid (3,2);
\end{tikzpicture}
```

12.3.4 Using Scopes Inside Paths

The `\path` command, which is described in much more detail in later sections, also takes graphic options. These options are local to the path. Furthermore, it is possible to create local scopes within a path simply by using curly braces as in



```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

Note that many options apply only to the path as a whole and cannot be scoped in this way. For example, it is not possible to scope the `color` of the path. See the explanations in the section on paths for more details.

Finally, certain elements that you specify in the argument to the `\path` command also take local options. For example, a node specification takes options. In this case, the options apply only to the node, not to the surrounding path.

12.4 Using Graphic Options

12.4.1 How Graphic Options Are Processed

Many commands and environments of Ti*k*Z accept *options*. These options are so-called *key lists*. To process the options, the following command is used, which you can also call yourself. Note that it is usually better not to call this command directly, since this will ensure that the effect of options are local to a well-defined scope.

`\tikzset{`⟨*options*⟩`}`

This command will process the ⟨*options*⟩ using the `\pgfkeys` command, documented in detail in Section 87, with the default path set to `/tikz`. Under normal circumstances, the ⟨*options*⟩ will be lists of comma-separated pairs of the form ⟨*key*⟩=⟨*value*⟩, but more fancy things can happen when you use the power of the `pgfkeys` mechanism, see Section 87 once more.

When a pair ⟨*key*⟩=⟨*value*⟩ is processed, the following happens:

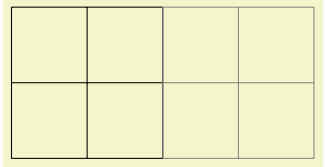
1. If the ⟨*key*⟩ is a full key (starts with a slash) it is handled directly as described in Section 87.
2. Otherwise (which is usually the case), it is checked whether `/tikz/`⟨*key*⟩ is a key and, if so, it is executed.
3. Otherwise, it is checked whether `/pgf/`⟨*key*⟩ is a key and, if so, it is executed.
4. Otherwise, it is checked whether ⟨*key*⟩ is a color and, if so, `color=`⟨*key*⟩ is executed.
5. Otherwise, it is checked whether ⟨*key*⟩ contains a dash and, if so, `arrows=`⟨*key*⟩ is executed.
6. Otherwise, it is checked whether ⟨*key*⟩ is the name of a shape and, if so, `shape=`⟨*key*⟩ is executed.
7. Otherwise, an error message is printed.

Note that by the above description, all keys starting with `/tikz` and also all keys starting with `/pgf` can be used as ⟨*key*⟩s in an ⟨*options*⟩ list.

12.4.2 Using Styles to Manage How Pictures Look

There is a way of organizing sets of graphic options "orthogonally" to the normal scoping mechanism. For example, you might wish all your "help lines" to be drawn in a certain way like, say, gray and thin (do *not* dash them, that distracts). For this, you can use *styles*.

A style is a key that, when used, causes a set of graphic options to be processed. Once a style has been defined, it can be used like any other key. For example, the predefined `help lines` style, which you should use for lines in the background like grid lines or construction lines.



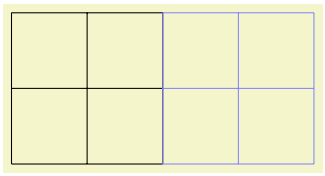
```
\begin{tikzpicture}
  \draw              (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

Defining styles is also done using options. Suppose we wish to define a style called `my style` and when this style is used, we want the draw color to be set to `red` and the fill color be set to `red!20`. To achieve this, we use the following option:

```
my style/.style={draw=red,fill=red!20}
```

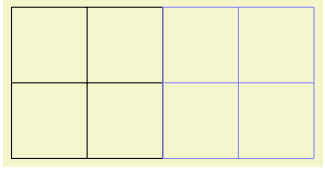
The meaning of the curious `/.style` is the following: "The key `my style` should not be used here but, rather, be defined. So, set up things such that using the key `my style` will, in the following, have the same effect as if we had written `draw=red,fill=red!20` instead."

Returning to the help lines example, suppose we prefer blue help lines. This could be achieved as follows:



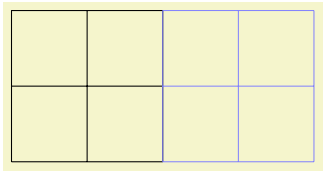
```
\begin{tikzpicture}[help lines/.style={blue!50,very thin}]
  \draw              (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

Naturally, one of the main ideas behind styles is that they can be used in different pictures. In this case, we have to use the `\tikzset` command somewhere at the beginning.



```
\tikzset{help lines/.style={blue!50,very thin}}
% ...
\begin{tikzpicture}
  \draw              (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

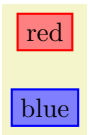
Since styles are just special cases of `pgfkeys`'s general style facility, you can actually do quite a bit more. Let us start with adding options to an already existing style. This is done using `/.append style` instead of `/.style`:



```
\begin{tikzpicture}[help lines/.append style=blue!50]
  \draw              (0,0) grid +(2,2);
  \draw[help lines] (2,0) grid +(2,2);
\end{tikzpicture}
```

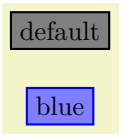
In the above example, the option `blue!50` is appended to the style `help lines`, which now has the same effect as `black!50,very thin,blue!50`. Note that two colors are set, so the last one will "win". There also exists a handler called `/.prefix style` that adds something at the beginning of the style.

Just as normal keys, styles can be parameterized. This means that you write ⟨*style*⟩=⟨*value*⟩ when you use the style instead of just ⟨*style*⟩. In this case, all occurrences of `#1` in ⟨*style*⟩ are replaced by ⟨*value*⟩. Here is an example that shows how this can be used.




```
\begin{tikzpicture}[outline/.style={draw=#1,thick,fill=#1!50}]
  \node [outline=red]  at (0,1) {red};
  \node [outline=blue] at (0,0) {blue};
\end{tikzpicture}
```

For parameterized styles you can also set a *default* value using the `/.default` handler:




```
\begin{tikzpicture}[outline/.style={draw=#1,thick,fill=#1!50},
                    outline/.default=black]
  \node [outline]      at (0,1) {default};
  \node [outline=blue] at (0,0) {blue};
\end{tikzpicture}
```

For more details on using and setting styles, see also Section 87.

13 Specifying Coordinates

13.1 Overview

A *coordinate* is a position on the canvas on which your picture is drawn. Ti*k*Z uses a special syntax for specifying coordinates. Coordinates are always put in round brackets. The general syntax is `([`⟨*options*⟩`]`⟨*coordinate specification*⟩`)`.

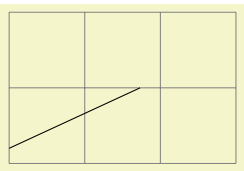
The ⟨*coordinate specification*⟩ specifies coordinates using one of many different possible *coordinate systems*. Examples are the Cartesian coordinate system or polar coordinates or spherical coordinates. No matter which coordinate system is used, in the end, a specific point on the canvas is represented by the coordinate.

There are two ways of specifying which coordinate system should be used:

Explicitly You can specify the coordinate system explicitly. To do so, you give the name of the coordinate system at the beginning, followed by `cs:`, which stands for "coordinate system", followed by a specification of the coordinate using the key–value syntax. Thus, the general syntax for ⟨*coordinate specification*⟩ in the explicit case is `(`⟨*coordinate system*⟩ `cs:`⟨*list of key–value pairs specific to the coordinate system*⟩`)`.

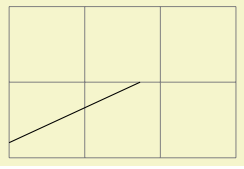
Implicitly The explicit specification is often too verbose when numerous coordinates should be given. Because of this, for the coordinate systems that you are likely to use often a special syntax is provided. Ti*k*Z will notice when you use a coordinate specified in a special syntax and will choose the correct coordinate system automatically.

Here is an example in which explicit the coordinate systems are specified explicitly:



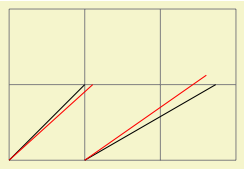
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (canvas cs:x=0cm,y=2mm)
     -- (canvas polar cs:radius=2cm,angle=30);
\end{tikzpicture}
```

In the next example, the coordinate systems are implicit:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0cm,2mm) -- (30:2cm);
\end{tikzpicture}
```

It is possible to give options that apply only to a single coordinate, although this makes sense for transformation options only. To give transformation options for a single coordinate, give these options at the beginning in brackets:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (1,1);
  \draw[red] (0,0) -- ([xshift=3pt] 1,1);
  \draw      (1,0) -- +(30:2cm);
  \draw[red] (1,0) -- +([shift=(135:5pt)] 30:2cm);
\end{tikzpicture}
```

13.2 Coordinate Systems

13.2.1 Canvas, XYZ, and Polar Coordinate Systems

Let us start with the basic coordinate systems.

Coordinate system `canvas`

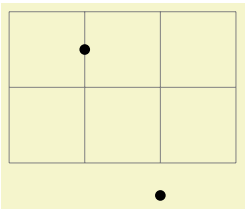
The simplest way of specifying a coordinate is to use the `canvas` coordinate system. You provide a dimension d_x using the `x=` option and another dimension d_y using the `y=` option. The position on the canvas is located at the position that is d_x to the right and d_y above the origin.

/tikz/cs/x=⟨*dimension*⟩ (no default, initially `0pt`)

Distance by which the coordinate is to the right of the origin. You can also write things like `1cm+2pt` since the mathematical engine is used to evaluate the ⟨*dimension*⟩.

/tikz/cs/y=⟨*dimension*⟩ (no default, initially `0pt`)

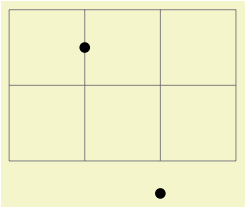
Distance by which the coordinate is above the origin.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \fill (canvas cs:x=1cm,y=1.5cm)    circle [radius=2pt];
  \fill (canvas cs:x=2cm,y=-5mm+2pt) circle [radius=2pt];
\end{tikzpicture}
```

To specify a coordinate in the coordinate system implicitly, you use two dimensions that are separated by a comma as in `(0cm,3pt)` or `(2cm,\textheight)`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \fill (1cm,1.5cm)    circle [radius=2pt];
  \fill (2cm,-5mm+2pt) circle [radius=2pt];
\end{tikzpicture}
```

Coordinate system xyz

The `xyz` coordinate system allows you to specify a point as a multiple of three vectors called the x-, y-, and z-vectors. By default, the x-vector points 1cm to the right, the y-vector points 1cm upwards, but this can be changed arbitrarily as explained in Section 25.2. The default z-vector points to $(-3.85\text{mm}, -3.85\text{mm})$.

To specify the factors by which the vectors should be multiplied before being added, you use the following three options:

/tikz/cs/x=⟨*factor*⟩ (no default, initially `0`)

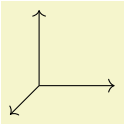
Factor by which the x-vector is multiplied.

/tikz/cs/y=⟨*factor*⟩ (no default, initially `0`)

Works like `x`.

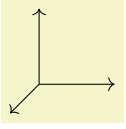
/tikz/cs/z=⟨*factor*⟩ (no default, initially `0`)

Works like `x`.



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (xyz cs:x=1);
  \draw (0,0) -- (xyz cs:y=1);
  \draw (0,0) -- (xyz cs:z=1);
\end{tikzpicture}
```

This coordinate system can also be selected implicitly. To do so, you just provide two or three comma-separated factors (not dimensions).



```
\begin{tikzpicture}[->]
  \draw (0,0) -- (1,0);
  \draw (0,0) -- (0,1,0);
  \draw (0,0) -- (0,0,1);
\end{tikzpicture}
```

Note: It is possible to use coordinates like `(1,2cm)`, which are neither `canvas` coordinates nor `xyz` coordinates. The rule is the following: If a coordinate is of the implicit form `(`⟨*x*⟩`,`⟨*y*⟩`)`, then ⟨*x*⟩ and ⟨*y*⟩ are checked, independently, whether they have a dimension or whether they are dimensionless. If both have a dimension, the `canvas` coordinate system is used. If both lack a dimension, the `xyz` coordinate system is used. If ⟨*x*⟩ has a dimension and ⟨*y*⟩ has not, then the sum of two coordinate `(`⟨*x*⟩`,0pt)` and `(0,`⟨*y*⟩`)` is used. If ⟨*y*⟩ has a dimension and ⟨*x*⟩ has not, then the sum of two coordinate `(`⟨*x*⟩`,0)` and `(0pt,`⟨*y*⟩`)` is used.

Note furthermore: An expression like `(2+3cm,0)` does *not* mean the same as `(2cm+3cm,0)`. Instead, if ⟨*x*⟩ or ⟨*y*⟩ internally uses a mixture of dimensions and dimensionless values, then all dimensionless values are "upgraded" to dimensions by interpreting them as `pt`. So, `2+3cm` is the same dimension as `2pt+3cm`.

Coordinate system `canvas polar`

The `canvas polar` coordinate system allows you to specify polar coordinates. You provide an angle using the `angle=` option and a radius using the `radius=` option. This yields the point on the canvas that is at the given radius distance from the origin at the given degree. An angle of zero degrees to the right, a degree of 90 upward.

`/tikz/cs/angle=`⟨*degrees*⟩ (no default)

The angle of the coordinate. The angle must always be given in degrees.

`/tikz/cs/radius=`⟨*dimension*⟩ (no default)

The distance from the origin.

`/tikz/cs/x radius=`⟨*dimension*⟩ (no default)

A polar coordinate is, after all, just a point on a circle of the given ⟨*radius*⟩. When you provide an x-radius and also a y-radius, you specify an ellipse instead of a circle. The `radius` option has the same effect as specifying identical `x radius` and `y radius` options.

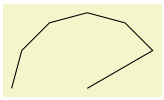
`/tikz/cs/y radius=`⟨*dimension*⟩ (no default)

Works like `x radius`.



```
\tikz \draw (0,0) -- (canvas polar cs:angle=30,radius=1cm);
```

The implicit form for canvas polar coordinates is the following: you specify the angle and the distance, separated by a colon as in `(30:1cm)`.



```
\tikz \draw    (0cm,0cm) -- (30:1cm) -- (60:1cm) -- (90:1cm)
           -- (120:1cm) -- (150:1cm) -- (180:1cm);
```

Two different radii are specified by writing `(30:1cm and 2cm)`.

For the implicit form, instead of an angle given as a number you can also use certain words. For example, `up` is the same as `90`, so that you can write `\tikz \draw (0,0) -- (2ex,0pt) -- +(up:1ex);` and get ⌴. Apart from `up` you can use `down`, `left`, `right`, `north`, `south`, `west`, `east`, `north east`, `north west`, `south east`, `south west`, all of which have their natural meaning.

Coordinate system `xyz polar`

This coordinate system work similarly to the `canvas polar` system. However, the radius and the angle are interpreted in the xy-coordinate system, not in the canvas system. More detailed, consider the circle or ellipse whose half axes are given by the current x-vector and the current y-vector. Then, consider the point that lies at a given angle on this ellipse, where an angle of zero is the same as the x-vector and an angle of 90 is the y-vector. Finally, multiply the resulting vector by the given radius factor. Voilà.

`/tikz/cs/angle=`⟨*degrees*⟩ (no default)

The angle of the coordinate interpreted in the ellipse whose axes are the x-vector and the y-vector.

`/tikz/cs/radius=`⟨*factor*⟩ (no default)

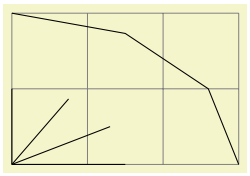
A factor by which the x-vector and y-vector are multiplied prior to forming the ellipse.

/tikz/cs/x radius=⟨*dimension*⟩ (no default)

A specific factor by which only the x-vector is multiplied.

/tikz/cs/y radius=⟨*dimension*⟩ (no default)

Works like `x radius`.

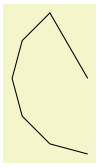


```
\begin{tikzpicture}[x=1.5cm,y=1cm]
  \draw[help lines] (0cm,0cm) grid (3cm,2cm);

  \draw (0,0) -- (xyz polar cs:angle=0,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=30,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=60,radius=1);
  \draw (0,0) -- (xyz polar cs:angle=90,radius=1);

  \draw (xyz polar cs:angle=0,radius=2)
     -- (xyz polar cs:angle=30,radius=2)
     -- (xyz polar cs:angle=60,radius=2)
     -- (xyz polar cs:angle=90,radius=2);
 \end{tikzpicture}
```

The implicit version of this option is the same as the implicit version of `canvas polar`, only you do not provide a unit.



```
\tikz[x={(0cm,1cm)},y={(-1cm,0cm)}]
  \draw  (0,0) -- (30:1) -- (60:1) -- (90:1)
               -- (120:1) -- (150:1) -- (180:1);
```

Coordinate system `xy polar`

This is just an alias for `xyz polar`, which some people might prefer as there is no z-coordinate involved in the `xyz polar` coordinates.

13.2.2 Barycentric Systems

In the barycentric coordinate system a point is expressed as the linear combination of multiple vectors. The idea is that you specify vectors v_1, v_2, …, v_n and numbers α_1, α_2, …, α_n. Then the barycentric coordinate specified by these vectors and numbers is

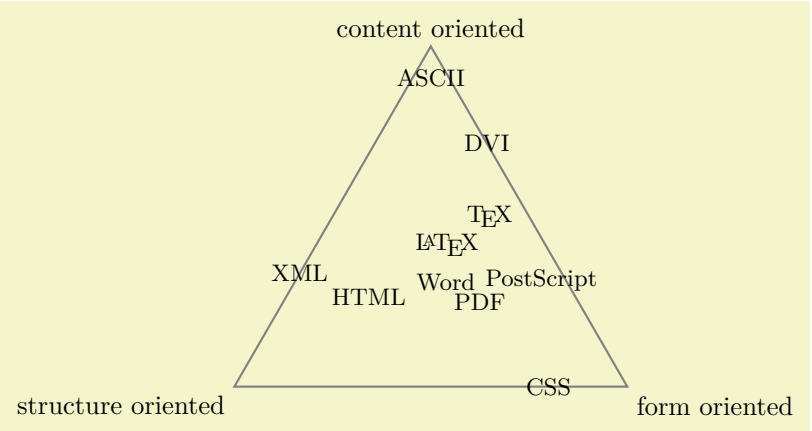
$$\frac{\alpha_1 v_1 + \alpha_2 v_2 + \cdots + \alpha_n v_n}{\alpha_1 + \alpha_2 + \cdots + \alpha_n}$$

The `barycentric cs` allows you to specify such coordinates easily.

Coordinate system `barycentric`

For this coordinate system, the ⟨*coordinate specification*⟩ should be a comma-separated list of expressions of the form ⟨*node name*⟩=⟨*number*⟩. Note that (currently) the list should not contain any spaces before or after the ⟨*node name*⟩ (unlike normal key–value pairs).

The specified coordinate is now computed as follows: Each pair provides one vector and a number. The vector is the `center` anchor of the ⟨*node name*⟩. The number is the ⟨*number*⟩. Note that (currently) you cannot specify a different anchor, so that in order to use, say, the `north` anchor of a node you first have to create a new coordinate at this north anchor. (Using for instance `\coordinate(mynorth) at (mynode.north);`.)




```
\begin{tikzpicture}
  \coordinate (content)   at (90:3cm);
  \coordinate (structure) at (210:3cm);
  \coordinate (form)      at (-30:3cm);

  \node [above]       at (content)   {content oriented};
  \node [below left]  at (structure) {structure oriented};
  \node [below right] at (form)      {form oriented};

  \draw [thick,gray] (content.south) -- (structure.north east) -- (form.north west) -- cycle;

  \small
  \node at (barycentric cs:content=0.5,structure=0.1 ,form=1)    {PostScript};
  \node at (barycentric cs:content=1  ,structure=0   ,form=0.4)  {DVI};
  \node at (barycentric cs:content=0.5,structure=0.5 ,form=1)    {PDF};
  \node at (barycentric cs:content=0  ,structure=0.25,form=1)    {CSS};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0)    {XML};
  \node at (barycentric cs:content=0.5,structure=1   ,form=0.4)  {HTML};
  \node at (barycentric cs:content=1  ,structure=0.2 ,form=0.8)  {\TeX};
  \node at (barycentric cs:content=1  ,structure=0.6 ,form=0.8)  {\LaTeX};
  \node at (barycentric cs:content=0.8,structure=0.8 ,form=1)    {Word};
  \node at (barycentric cs:content=1  ,structure=0.05,form=0.05) {ASCII};
\end{tikzpicture}
```

13.2.3 Node Coordinate System

In PGF and in Ti*k*Z it is quite easy to define a node that you wish to reference at a later point. Once you have defined a node, there are different ways of referencing points of the node. To do so, you use the following coordinate system:

Coordinate system `node`

This coordinate system is used to reference a specific point inside or on the border of a previously defined node. It can be used in different ways, so let us go over them one by one.

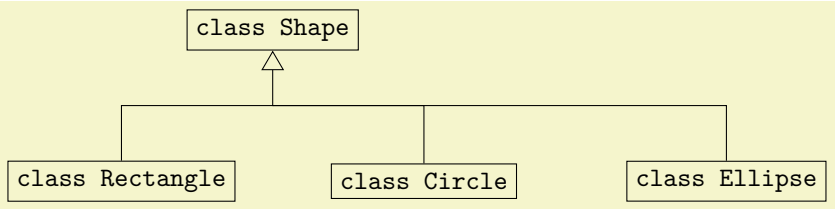
You can use three options to specify which coordinate you mean:

`/tikz/cs/name`=⟨*node name*⟩ (no default)

Specifies the node that you wish to use to specify a coordinate. The ⟨*node name*⟩ is the name that was previously used to name the node using the `name`=⟨*node name*⟩ option or the special node name syntax.

`/tikz/anchor`=⟨*anchor*⟩ (no default)

Specifies an anchor of the node. Here is an example:

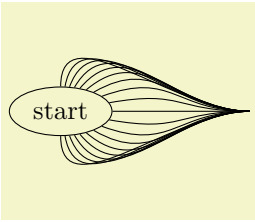



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[node font=\ttfamily]
  \node (shape)   at (0,2)  [draw] {class Shape};
  \node (rect)    at (-2,0) [draw] {class Rectangle};
  \node (circle)  at (2,0)  [draw] {class Circle};
  \node (ellipse) at (6,0)  [draw] {class Ellipse};

  \draw (node cs:name=circle,anchor=north) |- (0,1);
  \draw (node cs:name=ellipse,anchor=north) |- (0,1);
  \draw [arrows = -{Triangle[open, angle=60:3mm]}]
          (node cs:name=rect,anchor=north)
        |- (0,1) -| (node cs:name=shape,anchor=south);
\end{tikzpicture}
```

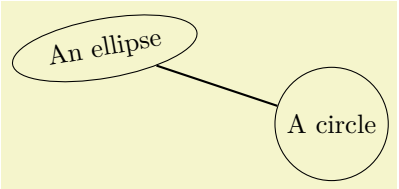
/tikz/cs/angle=⟨*degrees*⟩ (no default)

It is also possible to provide an angle *instead* of an anchor. This coordinate refers to a point of the node's border where a ray shot from the center in the given angle hits the border. Here is an example:




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \node (start) [draw,shape=ellipse] {start};
  \foreach \angle in {-90, -80, ..., 90}
    \draw (node cs:name=start,angle=\angle)
      .. controls +(\angle:1cm) and +(-1,0) .. (2.5,0);
  \end{tikzpicture}
```

It is possible to provide *neither* the `anchor=` option nor the `angle=` option. In this case, Ti*k*Z will calculate an appropriate border position for you. Here is an example:



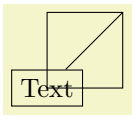

```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \path (0,0)  node(a) [ellipse,rotate=10,draw] {An ellipse}
        (3,-1) node(b) [circle,draw]            {A circle};
  \draw[thick] (node cs:name=a) -- (node cs:name=b);
\end{tikzpicture}
```

Ti*k*Z will be reasonably clever at determining the border points that you "mean", but, naturally, this may fail in some situations. If Ti*k*Z fails to determine an appropriate border point, the center will be used instead.

Automatic computation of anchors works only with the line-to operations `--`, the vertical/horizontal versions `|-` and `-|`, and with the curve-to operation `...` For other path commands, such as `parabola` or `plot`, the center will be used. If this is not desired, you should give a named anchor or an angle anchor.

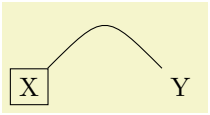
Note that if you use an automatic coordinate for both the start and the end of a line-to, as in `--(node cs:name=b)--`, then *two* border coordinates are computed with a move-to between them. This is usually exactly what you want.

If you use relative coordinates together with automatic anchor coordinates, the relative coordinates are computed relative to the node's center, not relative to the border point. Here is an example:




```
\tikz \draw (0,0) node(x) [draw] {Text}
            rectangle (1,1)
            (node cs:name=x) -- +(1,1);
```

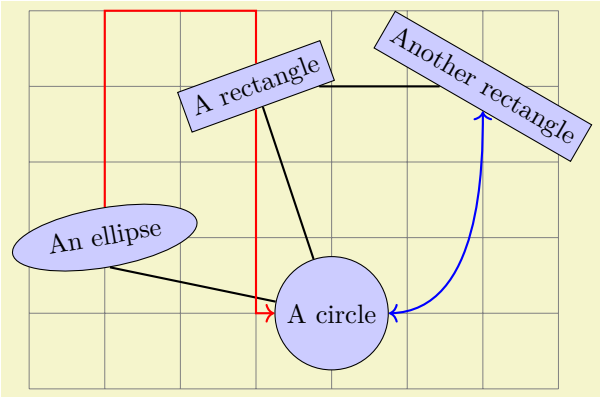
Similarly, in the following examples both control points are $(1,1)$:




```
\tikz \draw (0,0) node(x) [draw] {X}
            (2,0) node(y) {Y}
            (node cs:name=x) .. controls +(1,1) and +(-1,1) ..
            (node cs:name=y);
```

The implicit way of specifying the node coordinate system is to simply use the name of the node in parentheses as in `(a)` or to specify a name together with an anchor or an angle separated by a dot as in `(a.north)` or `(a.10)`.

Here is a more complete example:




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[fill=blue!20]
  \draw[help lines] (-1,-2) grid (6,3);
  \path (0,0)  node(a) [ellipse,rotate=10,draw,fill]    {An ellipse}
        (3,-1) node(b) [circle,draw,fill]               {A circle}
        (2,2)  node(c) [rectangle,rotate=20,draw,fill]  {A rectangle}
        (5,2)  node(d) [rectangle,rotate=-30,draw,fill] {Another rectangle};
  \draw[thick] (a.south) -- (b) -- (c) -- (d);
  \draw[thick,red,->] (a) |- +(1,3) -| (c) |- (b);
  \draw[thick,blue,<->] (b) .. controls +(right:2cm) and +(down:1cm) .. (d);
\end{tikzpicture}
```

13.2.4 Tangent Coordinate Systems

Coordinate system `tangent`

This coordinate system, which is available only when the Ti*k*Z library `calc` is loaded, allows you to compute the point that lies tangent to a shape. In detail, consider a ⟨*node*⟩ and a ⟨*point*⟩. Now, draw a straight line from the ⟨*point*⟩ so that it "touches" the ⟨*node*⟩ (more formally, so that it is *tangent* to this ⟨*node*⟩). The point where the line touches the shape is the point referred to by the `tangent` coordinate system.

The following options may be given:

`/tikz/cs/node=`⟨*node*⟩ (no default)

This key specifies the node on whose border the tangent should lie.

`/tikz/cs/point=`⟨*point*⟩ (no default)

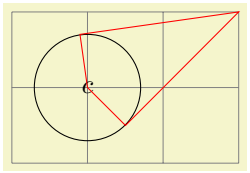
This key specifies the point through which the tangent should go.

/tikz/cs/solution=⟨*number*⟩ (no default)

Specifies which solution should be used if there are more than one.

A special algorithm is needed in order to compute the tangent for a given shape. Currently, tangents can be computed for nodes whose shape is one of the following:

- `coordinate`
- `circle`



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \coordinate (a) at (3,2);

  \node [circle,draw] (c) at (1,1) [minimum size=40pt] {$c$};

  \draw[red] (a)  -- (tangent cs:node=c,point={(a)},solution=1) --
       (c.center) -- (tangent cs:node=c,point={(a)},solution=2) -- cycle;
\end{tikzpicture}
```

There is no implicit syntax for this coordinate system.

13.2.5 Defining New Coordinate Systems

While the set of coordinate systems that Ti*k*Z can parse via their special syntax is fixed, it is possible and quite easy to define new explicitly named coordinate systems. For this, the following commands are used:

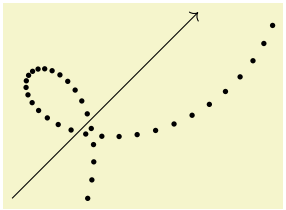
\tikzdeclarecoordinatesystem{⟨*name*⟩}{⟨*code*⟩}

This command declares a new coordinate system named ⟨*name*⟩ that can later on be used by writing (⟨*name*⟩ `cs:`⟨*arguments*⟩). When Ti*k*Z encounters a coordinate specified in this way, the ⟨*arguments*⟩ are passed to ⟨*code*⟩ as argument `#1`.

It is now the job of ⟨*code*⟩ to make sense of the ⟨*arguments*⟩. At the end of ⟨*code*⟩, the two TEX dimensions `\pgf@x` and `\pgf@y` should be have the x- and y-canvas coordinate of the coordinate.

It is not necessary, but customary, to parse ⟨*arguments*⟩ using the key–value syntax. However, you can also parse it in any way you like.

In the following example, a coordinate system `cylindrical` is defined.



```
\makeatletter
\define@key{cylindricalkeys}{angle}{\def\myangle{#1}}
\define@key{cylindricalkeys}{radius}{\def\myradius{#1}}
\define@key{cylindricalkeys}{z}{\def\myz{#1}}
\makeatother
\tikzdeclarecoordinatesystem{cylindrical}%
{%
  \setkeys{cylindricalkeys}{#1}%
  \pgfpointadd{\pgfpointxyz{0}{0}{\myz}}{\pgfpointpolarxy{\myangle}{\myradius}}
}
\begin{tikzpicture}[z=0.2pt]
  \draw [->] (0,0,0) -- (0,0,350);
  \foreach \num in {0,10,...,350}
    \fill (cylindrical cs:angle=\num,radius=1,z=\num) circle [radius=1pt];
\end{tikzpicture}
```

\tikzaliascoordinatesystem{⟨*new name*⟩}{⟨*old name*⟩}

Creates an alias of ⟨*old name*⟩.

13.3 Coordinates at Intersections

You will wish to compute the intersection of two paths. For the special and frequent case of two perpendicular lines, a special coordinate system called `perpendicular` is available. For more general cases, the `intersections` library can be used.

13.3.1 Intersections of Perpendicular Lines

A frequent special case of path intersections is the intersection of a vertical line going through a point p and a horizontal line going through some other point q. For this situation there is a useful coordinate system.

Coordinate system `perpendicular`

You can specify the two lines using the following keys:

`/tikz/cs/horizontal line through={(`⟨*coordinate*⟩`)}` (no default)

Specifies that one line is a horizontal line that goes through the given coordinate.

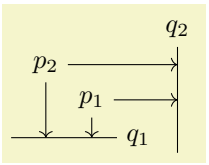
`/tikz/cs/vertical line through={(`⟨*coordinate*⟩`)}` (no default)

Specifies that the other line is vertical and goes through the given coordinate.

However, in almost all cases you should, instead, use the implicit syntax. Here, you write `(`⟨*p*⟩ `|-` ⟨*q*⟩`)` or `(`⟨*q*⟩ `-|` ⟨*p*⟩`)`.

For example, `(2,1 |- 3,4)` and `(3,4 -| 2,1)` both yield the same as `(2,4)` (provided the xy-coordinate system has not been modified).

The most useful application of the syntax is to draw a line up to some point on a vertical or horizontal line. Here is an example:



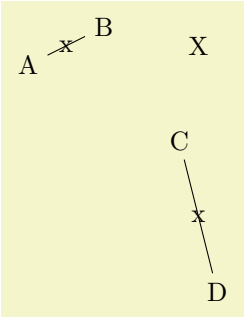

```
\begin{tikzpicture}
  \path (30:1cm) node(p1) {$p_1$}   (75:1cm) node(p2) {$p_2$};

  \draw (-0.2,0) -- (1.2,0) node(xline)[right] {$q_1$};
  \draw (2,-0.2) -- (2,1.2) node(yline)[above] {$q_2$};

  \draw[->] (p1) -- (p1 |- xline);
  \draw[->] (p2) -- (p2 |- xline);
  \draw[->] (p1) -- (p1 -| yline);
  \draw[->] (p2) -- (p2 -| yline);
\end{tikzpicture}
```

Note that in `(`⟨*c*⟩ `|-` ⟨*d*⟩`)` the coordinates ⟨*c*⟩ and ⟨*d*⟩ are *not* surrounded by parentheses. If they need to be complicated expressions (like a computation using the `$`-syntax), you must surround them with braces; parentheses will then be added around them.

As an example, let us specify a point that lies horizontally at the middle of the line from A to B and vertically at the middle of the line from C to D:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \node (A) at (0,1)    {A};
  \node (B) at (1,1.5)  {B};
  \node (C) at (2,0)    {C};
  \node (D) at (2.5,-2) {D};

  \draw (A) -- (B) node [midway] {x};
  \draw (C) -- (D) node [midway] {x};

  \node at ({$(A)!.5!(B)$} -| {$(C)!.5!(D)$}) {X};
\end{tikzpicture}
```

13.3.2 Intersections of Arbitrary Paths

TikZ Library `intersections`

```
\usetikzlibrary{intersections} % LaTeX and plain TeX
\usetikzlibrary[intersections] % ConTeXt
```

This library enables the calculation of intersections of two arbitrary paths. However, due to the low accuracy of TEX, the paths should not be "too complicated". In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

To find the intersections of two paths in Ti*k*Z, they must be "named". A "named path" is, quite simply, a path that has been named using the following key (note that this is a *different* key from the `name` key, which only attaches a hyperlink target to a path, but does not store the path in a way that is useful for the intersection computation):

`/tikz/name path=`⟨*name*⟩ (no default)
`/tikz/name path global=`⟨*name*⟩ (no default)

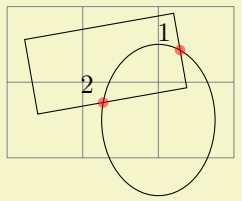
The effect of this key is that, after the path has been constructed, just before it is used, it is associated with ⟨*name*⟩. For `name path`, this association survives beyond the final semi-colon of the path but not the end of the surrounding scope. For `name path global`, the association will survive beyond any scope as well. Handle with care.

Any paths created by nodes on the (main) path are ignored, unless this key is explicitly used. If the same ⟨*name*⟩ is used for the main path and the node path(s), then the paths will be added together and then associated with ⟨*name*⟩.

To find the intersection of named paths, the following key is used:

`/tikz/name intersections={`⟨*options*⟩`}` (no default)

This key changes the key path to `/tikz/intersection` and processes ⟨*options*⟩. These options determine, among other things, which paths to use for the intersection. Having processed the options, any intersections are then found. A coordinate is created at each intersection, which by default, will be named `intersection-1`, `intersection-2`, and so on. Optionally, the prefix `intersection` can be changed, and the total number of intersections stored in a TEX-macro.




```
\usetikzlibrary {intersections}
\begin{tikzpicture}[every node/.style={opacity=1, black, above left}]
  \draw [help lines] grid (3,2);
  \draw [name path=ellipse] (2,0.5) ellipse [x radius=0.75cm, y radius=1cm];
  \draw [name path=rectangle, rotate=10] (0.5,0.5) rectangle +(2,1);
  \fill [red, opacity=0.5, name intersections={of=ellipse and rectangle}]
    (intersection-1) circle [radius=2pt] node {1}
    (intersection-2) circle [radius=2pt] node {2};
\end{tikzpicture}
```

The following keys can be used in ⟨*options*⟩:

`/tikz/intersection/of=`⟨*name path 1*⟩ `and` ⟨*name path 2*⟩ (no default)

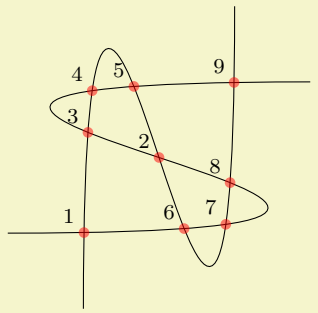
This key is used to specify the names of the paths to use for the intersection.

`/tikz/intersection/name=`⟨*prefix*⟩ (no default, initially `intersection`)

This key specifies the prefix name for the coordinate nodes placed at each intersection.

`/tikz/intersection/total=`⟨*macro*⟩ (no default)

This key means that the total number of intersections found will be stored in ⟨*macro*⟩.



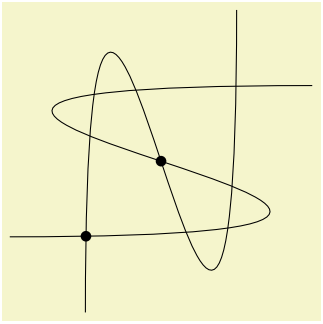

```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

  \fill [name intersections={of=curve 1 and curve 2, name=i, total=\t}]
        [red, opacity=0.5, every node/.style={above left, black, opacity=1}]
    \foreach \s in {1,...,\t}{(i-\s) circle [radius=2pt] node {\footnotesize\s}};
\end{tikzpicture}
```

`/tikz/intersection/by=`⟨*comma-separated list*⟩ (no default)

This key allows you to specify a list of names for the intersection coordinates. The intersection coordinates will still be named ⟨*prefix*⟩-⟨*number*⟩, but additionally the first coordinate will also

be named by the first element of the ⟨*comma-separated list*⟩. What happens is that the ⟨*comma-separated list*⟩ is passed to the `\foreach` statement and for ⟨*list member*⟩ a coordinate is created at the already-named intersection.

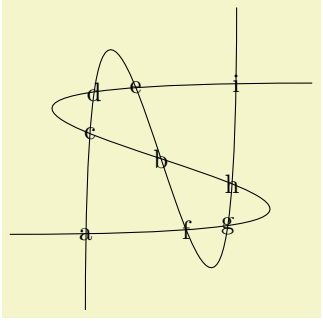


```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

  \fill [name intersections={of=curve 1 and curve 2, by={a,b}}]
        (a) circle [radius=2pt]
        (b) circle [radius=2pt];
\end{tikzpicture}
```

You can also use the `...` notation of the `\foreach` statement inside the ⟨*comma-separated list*⟩.

In case an element of the ⟨*comma-separated list*⟩ starts with options in square brackets, these options are used when the coordinate is created. A coordinate name can still, but need not, follow the options. This makes it easy to add labels to intersections:

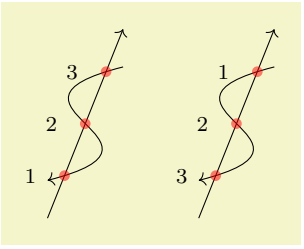



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \clip (-2,-2) rectangle (2,2);
  \draw [name path=curve 1] (-2,-1) .. controls (8,-1) and (-8,1) .. (2,1);
  \draw [name path=curve 2] (-1,-2) .. controls (-1,8) and (1,-8) .. (1,2);

  \fill [name intersections={
          of=curve 1 and curve 2,
          by={[label=center:a],[label=center:...],[label=center:i]}}];
\end{tikzpicture}
```

`/tikz/intersection/sort by=`⟨*path name*⟩ (no default)

By default, the intersections are simply returned in the order that the intersection algorithm finds them. Unfortunately, this is not necessarily a "helpful" ordering. This key can be used to sort the intersections along the path specified by ⟨*path name*⟩, which should be one of the paths mentioned in the `/tikz/intersection/of` key.

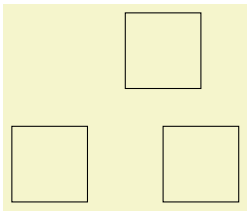



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
\clip (-0.5,-0.75) rectangle (3.25,2.25);
\foreach \pathname/\shift in {line/0cm, curve/2cm}{
  \tikzset{xshift=\shift}
  \draw [->, name path=curve] (1,1.5) .. controls (-1,1) and (2,0.5) .. (0,0);
  \draw [->, name path=line]  (0,-.5) -- (1,2) ;
  \fill [name intersections={of=line and curve,sort by=\pathname, name=i}]
    [red, opacity=0.5, every node/.style={left=.25cm, black, opacity=1}]
    \foreach \s in {1,2,3}{(i-\s) circle [radius=2pt] node {\footnotesize\s}};
}
\end{tikzpicture}
```

13.4 Relative and Incremental Coordinates

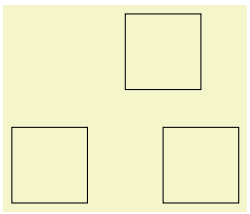
13.4.1 Specifying Relative Coordinates

You can prefix coordinates by `++` to make them "relative". A coordinate such as `++(1cm,0pt)` means "1cm to the right of the previous position, making this the new current position". Relative coordinates are often useful in "local" contexts:



```
\begin{tikzpicture}
  \draw (0,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (2,0)     -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
  \draw (1.5,1.5) -- ++(1,0) -- ++(0,1) -- ++(-1,0) -- cycle;
\end{tikzpicture}
```

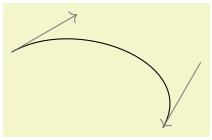
Instead of `++` you can also use a single `+`. This also specifies a relative coordinate, but it does not "update" the current point for subsequent usages of relative coordinates. Thus, you can use this notation to specify numerous points, all relative to the same "initial" point:



```
\begin{tikzpicture}
  \draw (0,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (2,0)     -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
  \draw (1.5,1.5) -- +(1,0) -- +(1,1) -- +(0,1) -- cycle;
\end{tikzpicture}
```

There is a special situation, where relative coordinates are interpreted differently. If you use a relative coordinate as a control point of a Bézier curve, the following rule applies: First, a relative first control point is taken relative to the beginning of the curve. Second, a relative second control point is taken relative to the end of the curve. Third, a relative end point of a curve is taken relative to the start of the curve.

This special behavior makes it easy to specify that a curve should "leave or arrive from a certain direction" at the start or end. In the following example, the curve "leaves" at 30° and "arrives" at 60°:



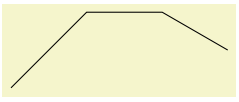
```
\begin{tikzpicture}
  \draw (1,0) .. controls +(30:1cm) and +(60:1cm) .. (3,-1);
  \draw[gray,->] (1,0) -- +(30:1cm);
  \draw[gray,<-] (3,-1) -- +(60:1cm);
\end{tikzpicture}
```

13.4.2 Rotational Relative Coordinates

You may sometimes wish to specify points relative not only to the previous point, but additionally relative to the tangent entering the previous point. For this, the following key is useful:

`/tikz/turn` (no value)

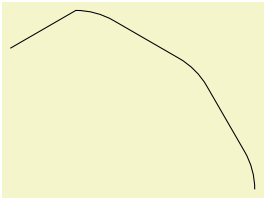
This key can be given as an option to a ⟨*coordinate*⟩ as in the following example:



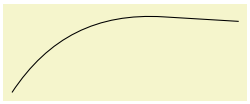
```
\tikz \draw (0,0) -- (1,1) -- ([turn]-45:1cm) -- ([turn]-30:1cm);
```

The effect of this key is to locally shift the coordinate system so that the last point reached is at the origin and the coordinate system is "turned" so that the x-axis points in the direction of a tangent entering the last point. This means, in effect, that when you use polar coordinates of the form ⟨*relative angle*⟩`:`⟨*distance*⟩ together with the `turn` option, you specify a point that lies at ⟨*distance*⟩ from the last point in the direction of the last tangent entering the last point, but with a rotation of ⟨*relative angle*⟩.

This key also works with curves …

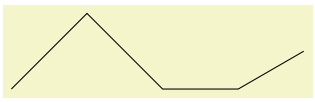


```
\tikz [delta angle=30, radius=1cm]
  \draw (0,0) arc [start angle=0]  -- ([turn]0:1cm)
              arc [start angle=30] -- ([turn]0:1cm)
              arc [start angle=60] -- ([turn]30:1cm);
```



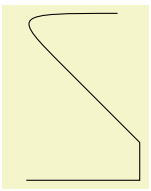
```
\tikz \draw (0,0) to [bend left] (2,1) -- ([turn]0:1cm);
```

...and with plots ...



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,0) } -- ([turn]30:1cm);
```

Although the above examples use polar coordinates with `turn`, you can also use any normal coordinate. For instance, `([turn]1,1)` will append a line of length $\sqrt{2}$ that is turns by 45° relative to the tangent to the last point.



```
\tikz \draw (0.5,0.5) -| (2,1) -- ([turn]1,1)
         .. controls ([turn]0:1cm) .. ([turn]-90:1cm);
```

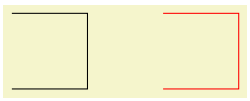
13.4.3 Relative Coordinates and Scopes

An interesting question is, how do relative coordinates behave in the presence of scopes? That is, suppose we use curly braces in a path to make part of it "local", how does that affect the current position? On the one hand, the current position certainly changes since the scope only affects options, not the path itself. On the other hand, it may be useful to "temporarily escape" from the updating of the current point.

Since both interpretations of how the current point and scopes should "interact" are useful, there is a (local!) option that allows you to decide which you need.

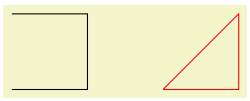
`/tikz/current point is local`=⟨*boolean*⟩ (no default, initially `false`)

Normally, the scope path operation has no effect on the current point. That is, curly braces on a path have no effect on the current position:



```
\begin{tikzpicture}
  \draw      (0,0) -- ++(1,0)   -- ++(0,1)   -- ++(-1,0);
  \draw[red] (2,0) -- ++(1,0) { -- ++(0,1) } -- ++(-1,0);
\end{tikzpicture}
```

If you set this key to `true`, this behavior changes. In this case, at the end of a group created on a path, the last current position reverts to whatever value it had at the beginning of the scope. More precisely, when Ti*k*Z encounters `}` on a path, it checks whether at this particular moment the key is set to `true`. If so, the current position reverts to the value it had when the matching `{` was read.



```
\begin{tikzpicture}
  \draw      (0,0) -- ++(1,0)   -- ++(0,1)   -- ++(-1,0);
  \draw[red] (2,0) -- ++(1,0)
     { [current point is local] -- ++(0,1) } -- ++(-1,0);
\end{tikzpicture}
```

In the above example, we could also have given the option outside the scope, for instance as a parameter to the whole scope.

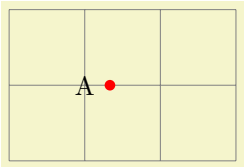
13.5 Coordinate Calculations

Ti*k*Z Library calc

```
\usetikzlibrary{calc} % LaTeX and plain TeX
\usetikzlibrary[calc] % ConTeXt
```

You need to load this library in order to use the coordinate calculation functions described in the present section.

It is possible to do some basic calculations that involve coordinates. In essence, you can add and subtract coordinates, scale them, compute midpoints, and do projections. For instance, `($(a) + 1/3*(1cm,0)$)` is the coordinate that is 1/3cm to the right of the point `a`:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \node (a) at (1,1) {A};
  \fill [red] ($(a) + 1/3*(1cm,0)$) circle [radius=2pt];
\end{tikzpicture}
```

13.5.1 The General Syntax

The general syntax is the following:

`([`⟨*options*⟩`]$`⟨*coordinate computation*⟩`$)`.

As you can see, the syntax uses the TEX math symbol `$` to indicate that a "mathematical computation" is involved. However, the `$` has no other effect, in particular, no mathematical text is typeset.

The ⟨*coordinate computation*⟩ has the following structure:

1. It starts with

   ⟨*factor*⟩`*`⟨*coordinate*⟩⟨*modifiers*⟩

2. This is optionally followed by `+` or `-` and then another

   ⟨*factor*⟩`*`⟨*coordinate*⟩⟨*modifiers*⟩

3. This is once more followed by `+` or `-` and another of the above modified coordinate; and so on.

In the following, the syntax of factors and of the different modifiers is explained in detail.

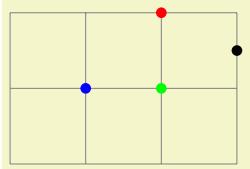
13.5.2 The Syntax of Factors

The ⟨*factor*⟩s are optional and detected by checking whether the ⟨*coordinate computation*⟩ starts with a `(`. Also, after each ± a ⟨*factor*⟩ is present if, and only if, the `+` or `-` sign is not directly followed by `(`.

If a ⟨*factor*⟩ is present, it is evaluated using the `\pgfmathparse` macro. This means that you can use pretty complicated computations inside a factor. A ⟨*factor*⟩ may even contain opening parentheses, which creates a complication: How does TikZ know where a ⟨*factor*⟩ ends and where a coordinate starts? For instance, if the beginning of a ⟨*coordinate computation*⟩ is `2*(3+4`…, it is not clear whether `3+4` is part of a ⟨*coordinate*⟩ or part of a ⟨*factor*⟩. Because of this, the following rule is used: Once it has been determined, that a ⟨*factor*⟩ is present, in principle, the ⟨*factor*⟩ contains everything up to the next occurrence of `*(`. Note that there is no space between the asterisk and the parenthesis.

It is permissible to put the ⟨*factor*⟩ in curly braces. This can be used whenever it is unclear where the ⟨*factor*⟩ would end.

Here are some examples of coordinate specifications that consist of exactly one ⟨*factor*⟩ and one ⟨*coordinate*⟩:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \fill [red] ($2*(1,1)$) circle [radius=2pt];
  \fill [green] (${1+1}*(1,.5)$) circle [radius=2pt];
  \fill [blue] ($cos(0)*sin(90)*(1,1)$) circle [radius=2pt];
  \fill [black] (${3*(4-3)}*(1,0.5)$) circle [radius=2pt];
\end{tikzpicture}
```

13.5.3 The Syntax of Partway Modifiers

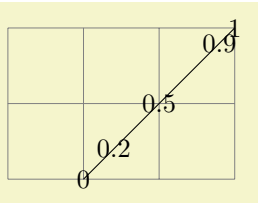
A ⟨*coordinate*⟩ can be followed by different ⟨*modifiers*⟩. The first kind of modifier is the *partway modifier*. The syntax (which is loosely inspired by Uwe Kern's `xcolor` package) is the following:

⟨*coordinate*⟩`!`⟨*number*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

One could write for instance

```
(1,2)!.75!(3,4)
```

The meaning of this is: "Use the coordinate that is three quarters on the way from `(1,2)` to `(3,4)`." In general, ⟨*coordinate x*⟩`!`⟨*number*⟩`!`⟨*coordinate y*⟩ yields the coordinate $(1 - \langle number\rangle)\langle coordinate\ x\rangle + \langle number\rangle\langle coordinate\ y\rangle$. Note that this is a bit different from the way the ⟨*number*⟩ is interpreted in the `xcolor` package: First, you use a factor between 0 and 1, not a percentage, and, second, as the ⟨*number*⟩ approaches 1, we approach the second coordinate, not the first. It is permissible to use a ⟨*number*⟩ that is smaller than 0 or larger than 1. The ⟨*number*⟩ is evaluated using the `\pgfmathparse` command and, thus, it can involve complicated computations.



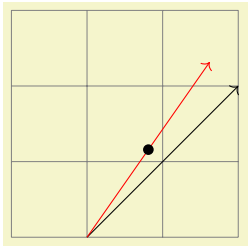

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw (1,0) -- (3,2);

  \foreach \i in {0,0.2,0.5,0.9,1}
    \node at ($(1,0)!\i!(3,2)$) {\i};
\end{tikzpicture}
```

The ⟨*second coordinate*⟩ may be prefixed by an ⟨*angle*⟩, separated with a colon, as in `(1,1)!.5!60:(2,2)`. The general meaning of ⟨*a*⟩`!`⟨*factor*⟩`!`⟨*angle*⟩`:`⟨*b*⟩ is: "First, consider the line from ⟨*a*⟩ to ⟨*b*⟩. Then rotate this line by ⟨*angle*⟩ *around the point* ⟨*a*⟩. Then the two endpoints of this line will be ⟨*a*⟩ and some point ⟨*c*⟩. Use this point ⟨*c*⟩ for the subsequent computation, namely the partway computation."

Here are two examples:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);

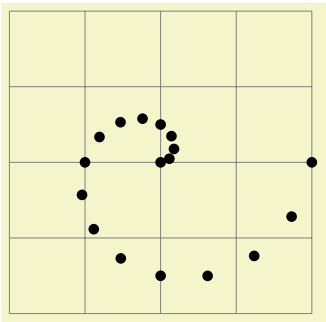
  \coordinate (a) at (1,0);
  \coordinate (b) at (3,2);

  \draw[->] (a) -- (b);

  \coordinate (c) at ($ (a)!1! 10:(b) $);

  \draw[->,red] (a) -- (c);

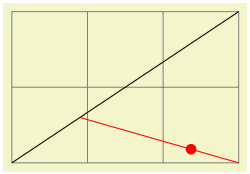
  \fill ($ (a)!.5! 10:(b) $) circle [radius=2pt];
\end{tikzpicture}
```



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (4,4);

  \foreach \i in {0,0.125,...,2}
    \fill ($(2,2) !\i! \i*180:(3,2)$) circle [radius=2pt];
\end{tikzpicture}
```

You can repeatedly apply modifiers. That is, after any modifier you can add another (possibly different) modifier.



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

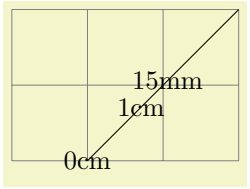
  \draw (0,0) -- (3,2);
  \draw[red] ($(0,0)!.3!(3,2)$) -- (3,0);
  \fill[red] ($(0,0)!.3!(3,2)!.7!(3,0)$) circle [radius=2pt];
\end{tikzpicture}
```

13.5.4 The Syntax of Distance Modifiers

A *distance modifier* has nearly the same syntax as a partway modifier, only you use a ⟨*dimension*⟩ (something like `1cm`) instead of a ⟨*factor*⟩ (something like `0.5`):

⟨*coordinate*⟩`!`⟨*dimension*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

When you write ⟨*a*⟩`!`⟨*dimension*⟩`!`⟨*b*⟩, this means the following: Use the point that is distanced ⟨*dimension*⟩ from ⟨*a*⟩ on the straight line from ⟨*a*⟩ to ⟨*b*⟩. Here is an example:



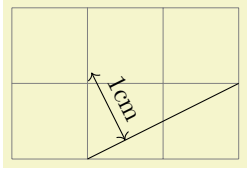

```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw (1,0) -- (3,2);

  \foreach \i in {0cm,1cm,15mm}
    \node at ($(1,0)!\i!(3,2)$) {\i};
\end{tikzpicture}
```

As before, if you use a ⟨*angle*⟩, the ⟨*second coordinate*⟩ is rotated by this much around the ⟨*coordinate*⟩ before it is used.

The combination of an ⟨*angle*⟩ of `90` degrees with a distance can be used to "offset" a point relative to a line. Suppose, for instance, that you have computed a point `(c)` that lies somewhere on a line from `(a)` to `(b)` and you now wish to offset this point by `1cm` so that the distance from this offset point to the line is `1cm`. This can be achieved as follows:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \coordinate (a) at (1,0);
  \coordinate (b) at (3,1);

  \draw (a) -- (b);

  \coordinate (c) at ($ (a)!.25!(b) $);
  \coordinate (d) at ($ (c)!1cm!90:(b) $);

  \draw [<->] (c) -- (d) node [sloped,midway,above] {1cm};
\end{tikzpicture}
```

13.5.5 The Syntax of Projection Modifiers

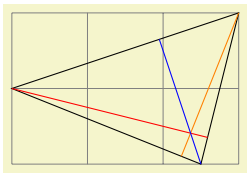
The projection modifier is also similar to the above modifiers: It also gives a point on a line from the ⟨*coordinate*⟩ to the ⟨*second coordinate*⟩. However, the ⟨*number*⟩ or ⟨*dimension*⟩ is replaced by a ⟨*projection coordinate*⟩:

⟨*coordinate*⟩`!`⟨*projection coordinate*⟩`!`⟨*angle*⟩`:`⟨*second coordinate*⟩

Here is an example:

```
(1,2)!(0,5)!(3,4)
```

The effect is the following: We project the ⟨*projection coordinate*⟩ orthogonally onto the line from ⟨*coordinate*⟩ to ⟨*second coordinate*⟩. This makes it easy to compute projected points:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \coordinate (a) at (0,1);
  \coordinate (b) at (3,2);
  \coordinate (c) at (2.5,0);

  \draw (a) -- (b) -- (c) -- cycle;

  \draw[red]    (a) -- ($(b)!(a)!(c)$);
  \draw[orange] (b) -- ($(a)!(b)!(c)$);
  \draw[blue]   (c) -- ($(a)!(c)!(b)$);
\end{tikzpicture}
```

14 Syntax for Path Specifications

A *path* is a series of straight and curved line segments. It is specified following a `\path` command and the specification must follow a special syntax, which is described in the subsections of the present section.

`\path`⟨*specification*⟩`;`

This command is available only inside a `{tikzpicture}` environment.

The ⟨*specification*⟩ is a long stream of *path operations*. Most of these path operations tell Ti*k*Z how the path is built. For example, when you write `--(0,0)`, you use a *line-to operation* and it means "continue the path from wherever you are to the origin".

At any point where Ti*k*Z expects a path operation, you can also give some graphic options, which is a list of options in brackets, such as `[rounded corners]`. These options can have different effects:

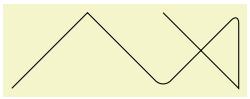
1. Some options take "immediate" effect and apply to all subsequent path operations on the path. For example, the `rounded corners` option will round all following corners, but not the corners "before" and if the `sharp corners` is given later on the path (in a new set of brackets), the rounding effect will end.



```
\tikz \draw (0,0) -- (1,1)
          [rounded corners] -- (2,0) -- (3,1)
          [sharp corners] -- (3,0) -- (2,1);
```

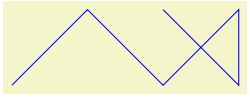
   Another example are the transformation options, which also apply only to subsequent coordinates.

2. The options that have immediate effect can be "scoped" by putting part of a path in curly braces. For example, the above example could also be written as follows:



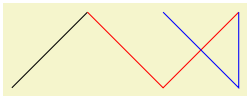
```
\tikz \draw (0,0) -- (1,1)
          {[rounded corners] -- (2,0) -- (3,1)}
          -- (3,0) -- (2,1);
```

3. Some options only apply to the path as a whole. For example, the `color=` option for determining the color used for, say, drawing the path always applies to all parts of the path. If several different colors are given for different parts of the path, only the last one (on the outermost scope) "wins":



```
\tikz \draw (0,0) -- (1,1)
          [color=red] -- (2,0) -- (3,1)
          [color=blue] -- (3,0) -- (2,1);
```

   Most options are of this type. In the above example, we would have had to "split up" the path into several `\path` commands:



```
\tikz{\draw (0,0) -- (1,1);
      \draw [color=red] (1,1) -- (2,0) -- (3,1);
      \draw [color=blue] (3,1) -- (3,0) -- (2,1);}
```

By default, the `\path` command does "nothing" with the path, it just "throws it away". Thus, if you write `\path(0,0)--(1,1);`, nothing is drawn in your picture. The only effect is that the area occupied by the picture is (possibly) enlarged so that the path fits inside the area. To actually "do" something with the path, an option like `draw` or `fill` must be given somewhere on the path. Commands like `\draw` do this implicitly.

Finally, it is also possible to give *node specifications* on a path. Such specifications can come at different locations, but they are always allowed when a normal path operation could follow. A node specification starts with `node`. Basically, the effect is to typeset the node's text as normal TEX text and to place it at the "current location" on the path. The details are explained in Section 17.

Note, however, that the nodes are *not* part of the path in any way. Rather, after everything has been done with the path what is specified by the path options (like filling and drawing the path due to a `fill` and a `draw` option somewhere in the ⟨*specification*⟩), the nodes are added in a post-processing step.

Note: When scanning for path operations TikZ expands tokens looking for valid path operations. This however implies that these tokens has to be fully expandable up to the point where it results in a valid path operation.

`/tikz/name`=⟨*path name*⟩ (no default)

Assigns a name to the path for reference (specifically, for reference in animations; for reference in intersections, use the `name path` command, which has a different purpose, see the `intersections` library for details). Since the name is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a path, but the name may *not* contain any punctuation like a dot, a comma, or a colon.

The following style influences scopes:

`/tikz/every path` (style, initially empty)

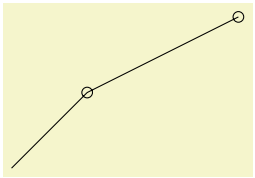
This style is installed at the beginning of every path. This can be useful for (temporarily) adding, say, the `draw` option to everything in a scope.



```
\begin{tikzpicture}
  [fill=yellow!80!black,      % only sets the color
   every path/.style={draw}]  % all paths are drawn
  \fill  (0,0) rectangle +(1,1);
  \shade (2,0) rectangle +(1,1);
\end{tikzpicture}
```

`/tikz/insert path`=⟨*path*⟩ (no default)

This key can be used inside an option to add something to the current path. This is mostly useful for defining styles that create graphic contents. This option should be used with care, for instance it should not be used as an argument of, say, a `node`. In the following example, we use a style to add little circles to a path.



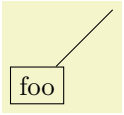
```
\tikz [c/.style={insert path={circle[radius=2pt]}}]
  \draw (0,0) -- (1,1) [c] -- (3,2) [c];
```

The effect is the same as of `(0,0) -- (1,1) circle[radius=2pt] -- (3,2) circle[radius=2pt]`.

The following options are for experts only:

`/tikz/append after command`=⟨*path*⟩ (no default)

Some of the path commands described in the following sections take optional arguments. For these commands, when you use this key inside these options, the ⟨*path*⟩ will be inserted *after* the path command is done. For instance, when you give this command in the option list of a node, the ⟨*path*⟩ will be added after the node. This is used by, for instance, the `label` option to allow you to specify a label in the option list of a node, but have this `label` cause a node to be added after another node.




```
\tikz \draw node [append after command={(foo)--(1,1)},draw] (foo){foo};
```

If this key is called multiple times, the effects accumulate, that is, all of the paths are added in the order to keys were found.

`/tikz/prefix after command`=⟨*path*⟩ (no default)

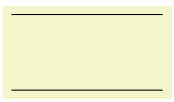
Works like `append after command`, only the accumulation order is inverse: The ⟨*path*⟩ is added before any earlier paths added using either `append after command` or `prefix after command`.

14.1 The Move-To Operation

The perhaps simplest operation is the move-to operation, which is specified by just giving a coordinate where a path operation is expected.

`\path` … ⟨*coordinate*⟩ … ;

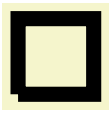
The move-to operation normally starts a path at a certain point. This does not cause a line segment to be created, but it specifies the starting point of the next segment. If a path is already under construction, that is, if several segments have already been created, a move-to operation will start a new part of the path that is not connected to any of the previous segments.



```
\begin{tikzpicture}
  \draw (0,0) --(2,0) (0,1) --(2,1);
\end{tikzpicture}
```

In the specification `(0,0) --(2,0) (0,1) --(2,1)` two move-to operations are specified: `(0,0)` and `(0,1)`. The other two operations, namely `--(2,0)` and `--(2,1)` are line-to operations, described next.

There is special coordinate called `current subpath start` that is always at the position of the last move-to operation on the current path.



```
\tikz [line width=2mm]
  \draw (0,0) -- (1,0) -- (1,1)
        -- (0,1) -- (current subpath start);
```

Note how in the above example the path is not closed (as `--cycle` would do). Rather, the line just starts and ends at the origin without being a closed path.

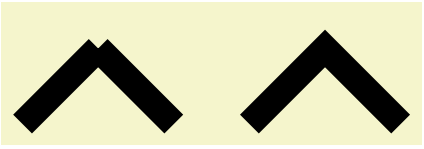
14.2 The Line-To Operation

14.2.1 Straight Lines

`\path` … `--`⟨*coordinate or cycle*⟩ … ;

The line-to operation extends the current path from the current point in a straight line to the given ⟨*coordinate*⟩ (the "or cycle" part is explained in a moment). The "current point" is the endpoint of the previous drawing operation or the point specified by a prior move-to operation.

When a line-to operation is used and some path segment has just been constructed, for example by another line-to operation, the two line segments become joined. This means that if they are drawn, the point where they meet is "joined" smoothly. To appreciate the difference, consider the following two examples: In the left example, the path consists of two path segments that are not joined, but they happen to share a point, while in the right example a smooth join is shown.



```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) --(1,1)  (1,1) --(2,0);
  \draw (3,0) -- (4,1) -- (5,0);
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

Instead of a coordinate following the two minus signs, you can also use the text `cycle`. This causes the straight line from the current point to go to the last point specified by a move-to operation. Note that this need not be the beginning of the path. Furthermore, a smooth join is created between the first segment created after the last move-to operation and the straight line appended by the cycle operation.

Consider the following example. In the left example, two triangles are created using three straight lines, but they are not joined at the ends. In the second example cycle operations are used.



```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) -- (1,0) -- (0,0) (2,0) -- (3,1) -- (3,0) -- (2,0);
  \draw (5,0) -- (6,1) -- (6,0) -- cycle (7,0) -- (8,1) -- (8,0) -- cycle;
  \useasboundingbox (0,1.5); % make bounding box higher
\end{tikzpicture}
```

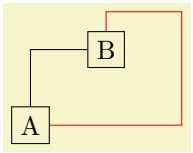
Writing `cycle` instead of a coordinate at the end of a path operation is possible with all path operations that end with a coordinate (such as `--` or `..` or `sin` or `grid`, but not `graph` or `plot`). In all cases, the effect is that the coordinate of the last moveto is used as the coordinate expected by the path operation and that a smooth join is added. (What actually happens is that the text `cycle` used with any path operation other than `--` gets replaced by `(current subpath start)--cycle`.)

14.2.2 Horizontal and Vertical Lines

Sometimes you want to connect two points via straight lines that are only horizontal and vertical. For this, you can use two path construction operations.

`\path` … `-|`⟨*coordinate or cycle*⟩ … ;

This operation means "first horizontal, then vertical".




```
\begin{tikzpicture}
  \draw (0,0) node(a) [draw] {A}  (1,1) node(b) [draw] {B};
  \draw (a.north) |- (b.west);
  \draw[color=red] (a.east) -| (2,1.5) -| (b.north);
\end{tikzpicture}
```

Instead of a coordinate you can also write `cycle` to close the path:



```
\begin{tikzpicture}[ultra thick]
  \draw (0,0) -- (1,1) -| cycle;
\end{tikzpicture}
```

`\path` … `|-`⟨*coordinate or cycle*⟩ … ;

This operation means "first vertical, then horizontal".

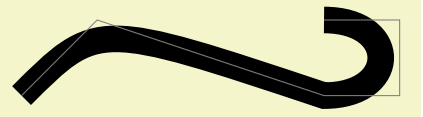
14.3 The Curve-To Operation

The curve-to operation allows you to extend a path using a Bézier curve.

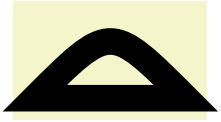
`\path` … `..controls`⟨*c*⟩`and`⟨*d*⟩`..`⟨*y or cycle*⟩ … ;

This operation extends the current path from the current point, let us call it x, via a curve to a point y (if, instead of a coordinate, you say `cycle` at the end, y will be the coordinate of the last move-to operation). The curve is a cubic Bézier curve. For such a curve, apart from y, you also specify two control points c and d. The idea is that the curve starts at x, "heading" in the direction of c. Mathematically spoken, the tangent of the curve at x goes through c. Similarly, the curve ends at y, "coming from" the other control point, d. The larger the distance between x and c and between d and y, the larger the curve will be.

If the "`and`⟨*d*⟩" part is not given, d is assumed to be equal to c.

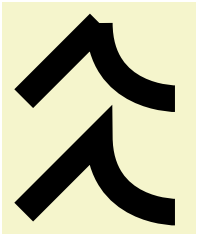


```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) .. controls (1,1) .. (4,0)
                         .. controls (5,0) and (5,1) .. (4,1);
  \draw[color=gray] (0,0) -- (1,1) -- (4,0) -- (5,0) -- (5,1) -- (4,1);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[line width=10pt] (0,0) -- (2,0) .. controls (1,1) .. cycle;
\end{tikzpicture}
```

As with the line-to operation, it makes a difference whether two curves are joined because they resulted from consecutive curve-to or line-to operations, or whether they just happen to have a common (end) point:



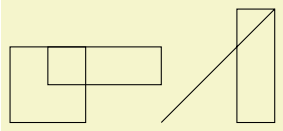
```
\begin{tikzpicture}[line width=10pt]
  \draw (0,0) -- (1,1) (1,1) .. controls (1,0) and (2,0) .. (2,0);
  \draw [yshift=-1.5cm]
        (0,0) -- (1,1)       .. controls (1,0) and (2,0) .. (2,0);
\end{tikzpicture}
```

14.4 The Rectangle Operation

A rectangle can obviously be created using four straight lines and a cycle operation. However, since rectangles are needed so often, a special syntax is available for them.

`\path` … `rectangle`⟨*corner or cycle*⟩ …;

When this operation is used, one corner will be the current point, another corner is given by ⟨*corner*⟩, which becomes the new current point.



```
\begin{tikzpicture}
  \draw (0,0) rectangle (1,1);
  \draw (.5,1) rectangle (2,0.5) (3,0) rectangle (3.5,1.5) -- (2,0);
\end{tikzpicture}
```

Just for consistency, you can also use `cycle` instead of a coordinate, but it is a bit unclear what use this might have.

14.5 Rounding Corners

All of the path construction operations mentioned up to now are influenced by the following option:

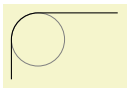
`/tikz/rounded corners=`⟨*inset*⟩ (default `4pt`)

When this option is in force, all corners (places where a line is continued either via line-to or a curve-to operation) are replaced by little arcs so that the corner becomes smooth.



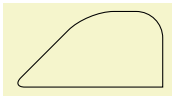
```
\tikz \draw [rounded corners] (0,0) -- (1,1)
          -- (2,0) .. controls (3,1) .. (4,0);
```

The ⟨*inset*⟩ describes how big the corner is. Note that the ⟨*inset*⟩ is *not* scaled if you use a scaling option like `scale=2`.



```
\begin{tikzpicture}
  \draw[color=gray,very thin] (10pt,15pt) circle[radius=10pt];
  \draw[rounded corners=10pt] (0,0) -- (0pt,25pt) -- (40pt,25pt);
\end{tikzpicture}
```

You can switch the rounded corners on and off "in the middle of path" and different corners in the same path can have different corner radii:



```
\begin{tikzpicture}
  \draw (0,0) [rounded corners=10pt] -- (1,1) -- (2,1)
                    [sharp corners] -- (2,0)
                [rounded corners=5pt] -- cycle;
\end{tikzpicture}
```

Here is a rectangle with rounded corners:



```
\tikz \draw[rounded corners=1ex] (0,0) rectangle (20pt,2ex);
```

You should be aware that there are several pitfalls when using this option. First, the rounded corner will only be an arc (part of a circle) if the angle is 90°. In other cases, the rounded corner will still be round, but "not as nice".

Second, if there are very short line segments in a path, the "rounding" may cause inadvertent effects. In such case it may be necessary to temporarily switch off the rounding using `sharp corners`.

`/tikz/sharp corners` (no value)

This options switches off any rounding on subsequent corners of the path.

14.6 The Circle and Ellipse Operations

Circles and ellipses are common path elements for which there is a special path operation.

`\path` … `circle[`⟨*options*⟩`]` …;

This command adds a circle to the current path where the center of the circle is the current point by default, but you can use the `at` option to change this. The new current point of the path will be (typically just remain) the center of the circle.

The radius of the circle is specified using the following options:

`/tikz/x radius=`⟨*value*⟩ (no default)

Sets the horizontal radius of the circle (which, when this value is different from the vertical radius, is actually an ellipse). The ⟨*value*⟩ may either be a dimension or a dimensionless number. In the latter case, the number is interpreted in the xy-coordinate system (if the x-unit is set to, say, `2cm`, then `x radius=3` will have the same effect as `x radius=6cm`).

`/tikz/y radius=`⟨*value*⟩ (no default)

Works like the `x radius`.

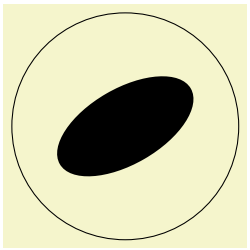
`/tikz/radius=`⟨*value*⟩ (no default)

Sets the `x radius` and `y radius` simultaneously.

`/tikz/at=`⟨*coordinate*⟩ (no default)

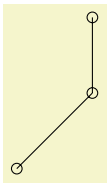
If this option is explicitly set inside the ⟨*options*⟩ (or indirectly via the `every circle` style), the ⟨*coordinate*⟩ is used as the center of the circle instead of the current point. Setting `at` to some value in an enclosing scope has no effect.

The ⟨*options*⟩ may also contain additional options like, say, a `rotate` or `scale`, that will only have an effect on the circle.



```
\begin{tikzpicture}
  \draw (1,0) circle [radius=1.5];
  \fill (1,0) circle [x radius=1cm, y radius=5mm, rotate=30];
\end{tikzpicture}
```

It is possible to set the `radius` also in some enclosing scope, in this case the options can be left out (but see the note below on what may follow):



```
\begin{tikzpicture}[radius=2pt]
  \draw (0,0) circle -- (1,1) circle -- ++(0,1) circle;
\end{tikzpicture}
```

The following style is used with every circle:

`/tikz/every circle` (style, no value)

You can use this key to set up, say, a default radius for every circle. The key will also be used with the `ellipse` operation.

In case you feel that the names `radius` and `x radius` are too long for your taste, you can easily create shorter aliases:

```
\tikzset{r/.style={radius=#1},rx/.style={x radius=#1},ry/.style={y radius=#1}}
```

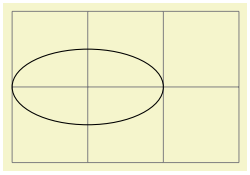
You can then say `circle [r=1cm]` or `circle [rx=1,ry=1.5]`. The reason Ti*k*Z uses the longer names by default is that it encourages people to write more readable code.

Note: There also exists an older syntax for circles, where the radius of the circle is given in parentheses right after the `circle` command as in `circle (1pt)`. Although this syntax is a bit more succinct, it is harder to understand for readers of the code and the use of parentheses for something other than a coordinate is ill-chosen.

Ti*k*Z will use the following rule to determine whether the old or the normal syntax is used: If `circle` is directly followed by something that (expands to) an opening parenthesis, then the old syntax is used and inside these following parentheses there must be a single number or dimension representing a radius. In all other cases the new syntax is used.

`\path` … `ellipse[`⟨*options*⟩`]` … `;`

This command has exactly the same effect as `circle`. The older syntax for this command is `ellipse (`⟨*x radius*⟩ `and` ⟨*y radius*⟩`)`. As for the `circle` command, this syntax is not as good as the standard syntax.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw (1,1) ellipse [x radius=1cm,y radius=.5cm];
\end{tikzpicture}
```

14.7 The Arc Operation

The *arc operation* allows you to add an arc to the current path.

`\path` … `arc[`⟨*options*⟩`]` … `;`

The `arc` operation adds a part of an ellipse to the current path. The radii of the ellipse are given by the values of `x radius` and `y radius`, which should be set in the ⟨*options*⟩. The arc will start at the current point and will end at the end of the arc. The arc will start and end at angles computed from the three keys `start angle`, `end angle`, and `delta angle`. Normally, the first two keys specify the start and end angle. However, in case one of them is empty, it is computed from the other key plus or minus the `delta angle`. In detail, if `end angle` is empty, it is set to the start angle plus the delta angle. If the start angle is missing, it is set to the end angle minus the delta angle. If all three keys are set, the delta angle is ignored.

`/tikz/start angle=`⟨*degrees*⟩ (no default)

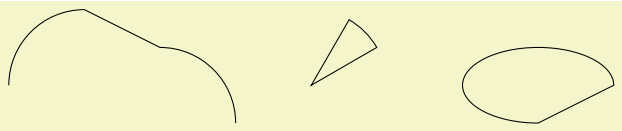
Sets the start angle.

`/tikz/end angle=`⟨*degrees*⟩ (no default)

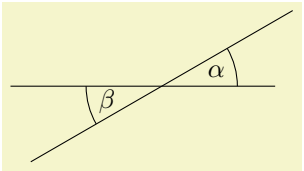
Sets the end angle.

`/tikz/delta angle=`⟨*degrees*⟩ (no default)

Sets the delta angle.



```
\begin{tikzpicture}[radius=1cm]
  \draw (0,0)  arc[start angle=180, end angle=90]
     -- (2,.5) arc[start angle=90,  delta angle=-90];
  \draw (4,0) -- +(30:1cm)
              arc [start angle=30,  delta angle=30] -- cycle;
  \draw (8,0) arc [start angle=0,   end angle=270,
                   x radius=1cm, y radius=5mm] -- cycle;
\end{tikzpicture}
```




```
\begin{tikzpicture}[radius=1cm,delta angle=30]
  \draw (-1,0) -- +(3.5,0);
  \draw (1,0) ++(210:2cm) -- +(30:4cm);
  \draw (1,0) +(0:1cm) arc [start angle=0];
  \draw (1,0) +(180:1cm) arc [start angle=180];
  \path (1,0) ++(15:.75cm) node{$\alpha$};
  \path (1,0) ++(15:-.75cm) node{$\beta$};
\end{tikzpicture}
```

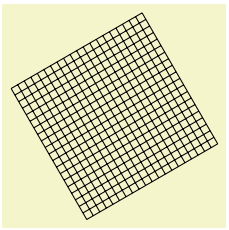
There also exists a shorter syntax for the arc operation, namely `arc` begin directly followed by `(`⟨*start angle*⟩`:`⟨*end angle*⟩`:`⟨*radius*⟩`)`. However, this syntax is harder to read, so the normal syntax should be preferred in general.

14.8 The Grid Operation

You can add a grid to the current path using the `grid` path operation.

`\path` … `grid[`⟨*options*⟩`]`⟨*corner or cycle*⟩ …`;`

This operation adds a grid filling a rectangle whose two corners are given by ⟨*corner*⟩ and by the previous coordinate. (Instead of a coordinate, you can also say `cycle` to use the position of the last move-to as the corner coordinate, but it is not very natural to do so.) Thus, the typical way in which a grid is drawn is `\draw (1,1) grid (3,3);`, which yields a grid filling the rectangle whose corners are at $(1, 1)$ and $(3, 3)$. All coordinate transformations apply to the grid.

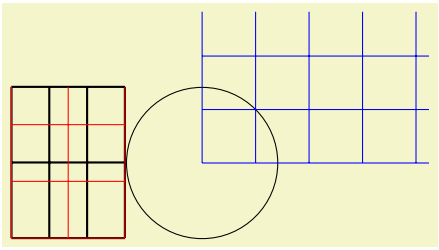


```
\tikz[rotate=30] \draw[step=1mm] (0,0) grid (2,2);
```

The ⟨*options*⟩, which are local to the `grid` operation, can be used to influence the appearance of the grid. The stepping of the grid is governed by the following options:

`/tikz/step`=⟨*number or dimension or coordinate*⟩ (no default, initially `1cm`)

Sets the stepping in both the x and y-directions. If a dimension is provided, this is used directly. If a number is provided, this number is interpreted in the xy-coordinate system. For example, if you provide the number 2, then the x-step is twice the x-vector and the y-step is twice the y-vector set by the `x=` and `y=` options. Finally, if you provide a coordinate, then the x-part of this coordinate will be used as the x-step and the y-part will be used as the y-coordinate.



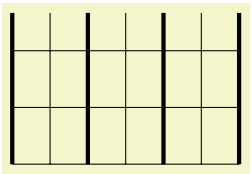
```
\begin{tikzpicture}[x=.5cm]
  \draw[thick] (0,0) grid [step=1]     (3,2);
  \draw[red]   (0,0) grid [step=.75cm] (3,2);
\end{tikzpicture}
\begin{tikzpicture}
  \draw        (0,0) circle [radius=1];
  \draw[blue]  (0,0) grid [step=(45:1)] (3,2);
\end{tikzpicture}
```

A complication arises when the x- and/or y-vector do not point along the axes. Because of this, the actual rule for computing the x-step and the y-step is the following: As the x- and y-steps, we use the x- and y-components or the following two vectors: The first vector is either $(\langle x\text{-}grid\text{-}step\text{-}number\rangle, 0)$ or $(\langle x\text{-}grid\text{-}step\text{-}dimension\rangle, 0\text{pt})$, the second vector is $(0, \langle y\text{-}grid\text{-}step\text{-}number\rangle)$ or $(0\text{pt}, \langle y\text{-}grid\text{-}step\text{-}dimension\rangle)$.

If the x-step or y-step is 0 or negative, the corresponding lines are not drawn.

`/tikz/xstep`=⟨*dimension or number*⟩ (no default, initially `1cm`)

Sets the stepping in the x-direction.



```
\begin{tikzpicture}
  \draw (0,0) grid [xstep=.5,ystep=.75] (3,2);
  \draw[ultra thick] (0,0) grid [ystep=0] (3,2);
\end{tikzpicture}
```

`/tikz/ystep`=⟨*dimension or number*⟩ (no default, initially `1cm`)

Sets the stepping in the y-direction.

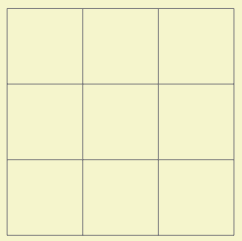
It is important to note that the grid is always "phased" such that it contains the point $(0, 0)$ if that point happens to be inside the rectangle. Thus, the grid does *not* always have an intersection at the corner points; this occurs only if the corner points are multiples of the stepping. Note that due to rounding

errors, the "last" lines of a grid may be omitted. In this case, you have to add an epsilon to the corner points.

The following style is useful for drawing grids:

`/tikz/help lines` (style, initially `line width=0.2pt,gray`)

This style makes lines "subdued" by using thin gray lines for them. However, this style is not installed automatically and you have to say, for example,



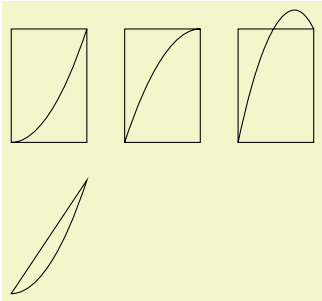
```
\tikz \draw[help lines] (0,0) grid (3,3);
```

14.9 The Parabola Operation

The `parabola` path operation continues the current path with a parabola. A parabola is a (shifted and scaled) curve defined by the equation $f(x) = x^2$ and looks like this: $\cup$.

`\path` … `parabola[`⟨*options*⟩`]bend`⟨*bend coordinate*⟩⟨*coordinate or cycle*⟩ … ;

This operation adds a parabola through the current point and the given ⟨*coordinate*⟩ or, if `cycle` is used instead of coordinate at the end, the ⟨*coordinate*⟩ is set to the position of the last move-to and the path gets closed after the parabola. If the `bend` is given, it specifies where the bend should go; the ⟨*options*⟩ can also be used to specify where the bend is. By default, the bend is at the old current point.



```
\begin{tikzpicture}
  \draw                 (0,0) rectangle                (1,1.5)
                        (0,0) parabola                 (1,1.5);
  \draw[xshift=1.5cm]   (0,0) rectangle                (1,1.5)
                        (0,0) parabola[bend at end]    (1,1.5);
  \draw[xshift=3cm]     (0,0) rectangle                (1,1.5)
                        (0,0) parabola bend (.75,1.75) (1,1.5);

  \draw[yshift=-2cm]    (1,1.5) --
                        (0,0) parabola                 cycle;
\end{tikzpicture}
```

The following options influence parabolas:

`/tikz/bend=`⟨*coordinate*⟩ (no default)

Has the same effect as saying `bend`⟨*coordinate*⟩ outside the ⟨*options*⟩. The option specifies that the bend of the parabola should be at the given ⟨*coordinate*⟩. You have to take care yourself that the bend position is a "valid" position; which means that if there is no parabola of the form $f(x) = ax^2 + bx + c$ that goes through the old current point, the given bend, and the new current point, the result will not be a parabola.

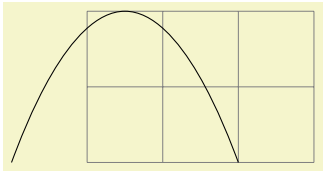
There is one special property of the ⟨*coordinate*⟩: When a relative coordinate is given like `+(0,0)`, the position relative to this coordinate is "flexible". More precisely, this position lies somewhere on a line from the old current point to the new current point. The exact position depends on the next option.

`/tikz/bend pos=`⟨*fraction*⟩ (no default)

Specifies where the "previous" point is relative to which the bend is calculated. The previous point will be at the ⟨*fraction*⟩th part of the line from the old current point to the new current point.

The idea is the following: If you say `bend pos=0` and `bend +(0,0)`, the bend will be at the old current point. If you say `bend pos=1` and `bend +(0,0)`, the bend will be at the new current point.

If you say `bend pos=0.5` and `bend +(0,2cm)`, the bend will be 2cm above the middle of the line between the start and end point. This is most useful in situations such as the following:

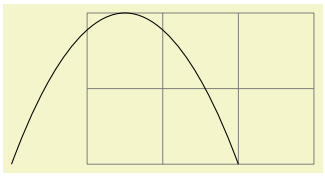


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[bend pos=0.5] bend +(0,2) +(3,0);
\end{tikzpicture}
```

In the above example, the `bend +(0,2)` essentially means "a parabola that is 2cm high" and `+(3,0)` means "and 3cm wide". Since this situation arises often, there is a special shortcut option:

`/tikz/parabola height=`⟨*dimension*⟩ (no default)

This option has the same effect as `[bend pos=0.5,bend={+(0pt,`⟨*dimension*⟩`)}]`.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (-1,0) parabola[parabola height=2cm] +(3,0);
\end{tikzpicture}
```

The following styles are useful shortcuts:

`/tikz/bend at start` (style, no value)

This places the bend at the start of a parabola. It is a shortcut for the following options: `bend pos=0,bend={+(0,0)}`.

`/tikz/bend at end` (style, no value)

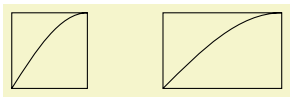
This places the bend at the end of a parabola.

14.10 The Sine and Cosine Operation

The `sin` and `cos` operations are similar to the `parabola` operation. They, too, can be used to draw (parts of) a sine or cosine curve.

`\path` … `sin`⟨*coordinate or cycle*⟩ … ;

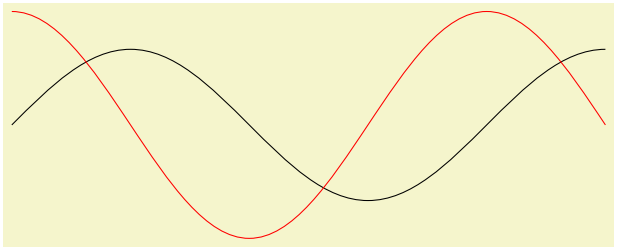
The effect of `sin` is to draw a scaled and shifted version of a sine curve in the interval $[0, \pi/2]$. The scaling and shifting is done in such a way that the start of the sine curve in the interval is at the old current point and the end of the curve in the interval is at ⟨*coordinate*⟩. Here is an example that should clarify this:



```
\tikz \draw (0,0) rectangle (1,1)     (0,0) sin (1,1)
            (2,0) rectangle +(1.57,1) (2,0) sin +(1.57,1);
```

`\path` … `cos`⟨*coordinate or cycle*⟩ … ;

This operation works similarly, only a cosine in the interval $[0, \pi/2]$ is drawn. By correctly alternating `sin` and `cos` operations, you can create a complete sine or cosine curve:



```
\begin{tikzpicture}[xscale=1.57]
  \draw (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0) sin (5,1);
  \draw[color=red] (0,1.5) cos (1,0) sin (2,-1.5) cos (3,0) sin (4,1.5) cos (5,0);
\end{tikzpicture}
```

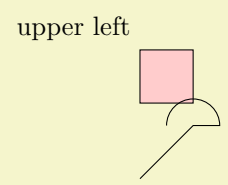
Note that there is no way to (conveniently) draw an interval on a sine or cosine curve whose end points are not multiples of $\pi/2$.

14.11 The SVG Operation

The `svg` operation can be used to extend the current path by a path given in the SVG path data syntax. This syntax is described in detail in Section 8.3 of the SVG 1.1 specification, please consult this specification for details.

`\path` … `svg[`⟨*options*⟩`]{`⟨*path data*⟩`}` … `;`

This operation adds the path specified in the ⟨*path data*⟩ in SVG 1.1 PATH DATA syntax to the current path. Unlike the SVG-specification, it *is* permissible that the path data does not start with a move-to command (`m` or `M`), in which case the last point of the current path is used as start point. The optional ⟨*options*⟩ apply locally to this path operation, typically you will use them to set up, say, some transformations.




```
\usetikzlibrary {svg.path}
\begin{tikzpicture}
  \filldraw [fill=red!20] (0,1) svg[scale=2] {h 10 v 10 h -10}
    node [above left] {upper left} -- cycle;

  \draw svg {M 0 0 L 20 20 h 10 a 10 10 0 0 0 -20 0};
\end{tikzpicture}
```

An SVG coordinate like `10 20` is always interpreted as `(10pt,20pt)`, so the basic unit is always points (`pt`). The *xy*-coordinate system is not used. However, you can use scaling to (locally) change the basic unit. For instance, `svg[scale=1cm]` (yes, this works, although some rather evil magic is involved) will cause 1cm to be the basic unit.

Instead of curly braces, you can also use quotation marks to indicate the start and end of the SVG path.

Warning: The arc operations (`a` and `A`) are numerically instable. This means that they will be quite imprecise, except when the angle is a multiple of 90° (as is, fortunately, most often the case).

14.12 The Plot Operation

The `plot` operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

Since the syntax and the behavior of this command are a bit complex, they are described in the separated Section 22.

14.13 The To Path Operation

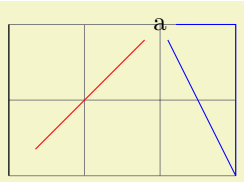
The `to` operation is used to add a user-defined path from the previous coordinate to the following coordinate. When you write `(a) to (b)`, a straight line is added from `a` to `b`, exactly as if you had written `(a) -- (b)`. However, if you write `(a) to [out=135,in=45] (b)` a curve is added to the path, which leaves at an angle of 135° at `a` and arrives at an angle of 45° at `b`. This is because the options `in` and `out` trigger a special path to be used instead of the straight line.

`\path` … `to[`⟨*options*⟩`]` ⟨*nodes*⟩ ⟨*coordinate or cycle*⟩ … `;`

This path operation inserts the path currently set via the `to path` option at the current position. The ⟨*options*⟩ can be used to modify (perhaps implicitly) the `to path` and to set up how the path will be rendered.

Before the `to path` is inserted, a number of macros are set up that can "help" the `to path`. These are `\tikztostart`, `\tikztotarget`, and `\tikztonodes`; they are explained in the following.

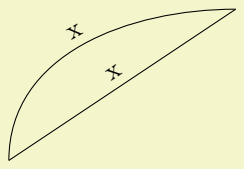
Start and Target Coordinates. The `to` operation is always followed by a ⟨*coordinate*⟩, called the target coordinate, or the text `cycle`, in which case the last move-to is used as a coordinate and the path gets closed. The macro `\tikztotarget` is set to this coordinate (without its parentheses). There is also a *start coordinate*, which is the coordinate preceding the `to` operation. This coordinate can be accessed via the macro `\tikztostart`. In the following example, for the first `to`, the macro `\tikztostart` is `0pt,0pt` and the `\tikztotarget` is `0,2`. For the second `to`, the macro `\tikztostart` is `10pt,10pt` and `\tikztotarget` is `a`. For the third, they are set to `a` and `current subpath start`.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \node       (a)         at (2,2) {a};

  \draw       (0,0)       to (0,2);
  \draw[red]  (10pt,10pt) to (a);
  \draw[blue] (3,0) -- (3,2) -- (a) to cycle;
\end{tikzpicture}
```

Nodes on to–paths. It is possible to add nodes to the paths constructed by a `to` operation. To do so, you specify the nodes between the `to` keyword and the coordinate (if there are options to the `to` operation, these come first). The effect of `(a) to node {x} (b)` (typically) is the same as if you had written `(a) -- node {x} (b)`, namely that the node is placed on the `to`. This can be used to add labels to `to`s:



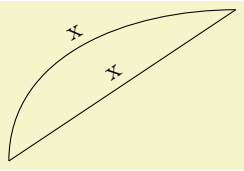

```
\begin{tikzpicture}
  \draw (0,0) to node [sloped,above] {x} (3,2);

  \draw (0,0) to[out=90,in=180] node [sloped,above] {x} (3,2);
\end{tikzpicture}
```

Instead of writing the node between the `to` keyword and the target coordinate, you may also use the following keys to create such nodes:

`/tikz/edge node=`⟨*node specification*⟩ (no default)

This key can be used inside the ⟨*options*⟩ of a `to` path command. It will add the ⟨*node specification*⟩ to the list of nodes to be placed on the connecting line, just as if you had written the ⟨*node specification*⟩ directly after the `to` keyword:



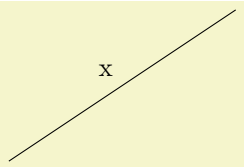

```
\begin{tikzpicture}
  \draw (0,0) to [edge node={node [sloped,above] {x}}] (3,2);

  \draw (0,0) to [out=90,in=180,
                  edge node={node [sloped,above] {x}}] (3,2);
\end{tikzpicture}
```

This key is mostly useful to create labels automatically using other keys.

`/tikz/edge label=`⟨*text*⟩ (no default)

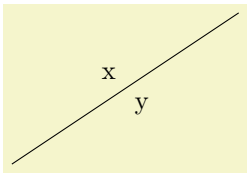
A shorthand for `edge node={node[auto]{`⟨*text*⟩`}}`.




```
\tikz \draw (0,0) to [edge label=x] (3,2);
```

`/tikz/edge label'=`⟨*text*⟩ (no default)

A shorthand for `edge node={node[auto,swap]{`⟨*text*⟩`}}`.



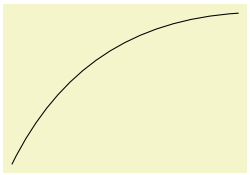

```
\tikz \draw (0,0) to [edge label=x, edge label'=y] (3,2);
```

When the `quotes` library is loaded, additional ways of specifying nodes on to–paths become available, see Section 17.12.2.

Styles for to-paths. In addition to the ⟨*options*⟩ given after the `to` operation, the following style is also set at the beginning of the to path:

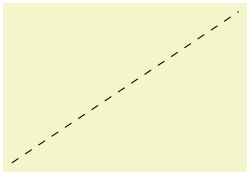
`/tikz/every to` (style, initially empty)

This style is installed at the beginning of every to.



```
\tikz[every to/.style={bend left}]
  \draw (0,0) to (3,2);
```

Note that, as explained below, every to path is implicitly surrounded by curly braces. This means that options like `draw` given in an `every to` do not actually influence the path. You can fix this by using the `append after command` option:



```
\tikz[every to/.style={append after command={[draw,dashed]}}]
  \draw (0,0) to (3,2);
```

Options. The ⟨*options*⟩ given with the `to` allow you to influence the appearance of the `to path`. Mostly, these options are used to change the `to path`. This can be used to change the path from a straight line to, say, a curve.

The path used is set using the following option:

`/tikz/to path`=⟨*path*⟩ (no default)

Whenever a `to` operation is used, the ⟨*path*⟩ is inserted. More precisely, the following path is added:

```
{[every to,⟨options⟩] ⟨path⟩ }
```

The ⟨*options*⟩ are the options given to the `to` operation, the ⟨*path*⟩ is the path set by this option `to path`.

Inside the ⟨*path*⟩, different macros are used to reference the from- and to-coordinates. In detail, these are:

- `\tikztostart` will expand to the from-coordinate (without the parentheses).
- `\tikztotarget` will expand to the to-coordinate.
- `\tikztonodes` will expand to the nodes between the `to` operation and the coordinate. Furthermore, these nodes will have the `pos` option set implicitly.

Let us have a look at a simple example. The standard straight line for a `to` is achieved by the following ⟨*path*⟩:

```
-- (\tikztotarget) \tikztonodes
```

Indeed, this is the default setting for the path. When we write `(a) to (b)`, the ⟨*path*⟩ will expand to `(a) -- (b)`, when we write

```
(a) to[red] node {x} (b)
```

the ⟨*path*⟩ will expand to

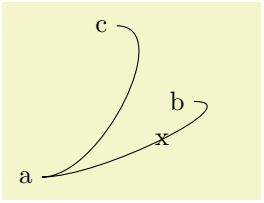
```
(a) -- (b) node[red] {x}
```

It is not possible to specify the path

```
-- \tikztonodes (\tikztotarget)
```

since Ti*k*Z does not allow one to have a macro after `--` that expands to a node.

Now let us have a look at how we can modify the ⟨*path*⟩ sensibly. The simplest way is to use a curve.

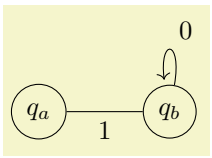



```
\begin{tikzpicture}[to path={
    .. controls +(1,0) and +(1,0) .. (\tikztotarget) \tikztonodes}]

  \node (a) at (0,0) {a};
  \node (b) at (2,1) {b};
  \node (c) at (1,2) {c};

  \draw (a) to node {x} (b)
        (a) to          (c);
\end{tikzpicture}
```

Here is another example:




```
\tikzset{
  my loop/.style={to path={
    .. controls +(80:1) and +(100:1) .. (\tikztotarget) \tikztonodes}},
  my state/.style={circle,draw}}

\begin{tikzpicture}[shorten >=2pt]
  \node [my state] (a) at (210:1) {$q_a$};
  \node [my state] (b) at (330:1) {$q_b$};

  \draw[->] (a) to           node[below]       {1} (b)
                to [my loop] node[above right] {0} (b);
\end{tikzpicture}
```

`/tikz/execute at begin to=`⟨*code*⟩ (no default)

The ⟨*code*⟩ is executed prior to the `to`. This can be used to draw one or more additional paths or to do additional computations.

`/tikz/execute at end to=`⟨*code*⟩ (no default)

Works like the previous option, only this code is executed after the to path has been added.

`/tikz/every to` (style, initially empty)

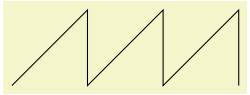
This style is installed at the beginning of every to.

There are a number of predefined `to path`s, see Section 74 for a reference.

14.14 The Foreach Operation

`\path` … `foreach`⟨*variables*⟩`[`⟨*options*⟩`]` `in` ⟨*list*⟩ `{`⟨*path commands*⟩`}` … `;`

The `foreach` operation can be used to repeatedly insert the ⟨*path commands*⟩ into the current path. Naturally, the ⟨*path commands*⟩ should internally reference some of the ⟨*variables*⟩ so that you do not insert exactly the same path repeatedly, but rather variations. For historical reasons, you can also write `\foreach` instead of `foreach`.



```
\tikz \draw (0,0) foreach \x in {1,...,3} { -- (\x,1) -- (\x,0) };
```

See Section 88 for more details on the for-each-command.

14.15 The Let Operation

The *let operation* is the first of a number of path operations that do not actually extend that path, but have different, mostly local, effects. It requires the `calc` library, see Section 13.5.

`\path` … `let`⟨*assignment*⟩ `,`⟨*assignment*⟩`,`⟨*assignment*⟩… `in` …;

When this path operation is encountered, the ⟨*assignment*⟩s are evaluated, one by one. This will store coordinate and number in special *registers* (which are local to Ti*k*Z, they have nothing to do with TEX registers). Subsequently, one can access the contents of these registers using the macros `\p`, `\x`, `\y`, and `\n`.

The first kind of permissible ⟨*assignment*⟩s have the following form:

`\n`⟨*number register*⟩`={`⟨*formula*⟩`}`

When an assignment has this form, the ⟨*formula*⟩ is evaluated using the `\pgfmathparse` operation. The result is stored in the ⟨*number register*⟩. If the ⟨*formula*⟩ involves a dimension anywhere (as in `2*3cm/2`), then the ⟨*number register*⟩ stores the resulting dimension with a trailing `pt`. A ⟨*number register*⟩ can be named arbitrarily and is a normal TEX parameter to the `\n` macro. Possible names are `{left corner}`, but also just a single digit like `5`.

Let us call the path that follows a let operation its *body*. Inside the body, the `\n` macro can be used to access the register.

`\n{`⟨*number register*⟩`}`

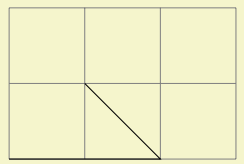
When this macro is used on the left-hand side of an `=`-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an `=`-sign or in the body of the let operation, then it expands to the value stored in the ⟨*number register*⟩. This will either be a dimensionless number like `2.0` or a dimension like `5.6pt`.

For instance, if we say `let \n1={1pt+2pt}, \n2={1+2} in ...`, then inside the `...` part the macro `\n1` will expand to `3pt` and `\n2` expands to `3`.

The second kind of ⟨*assignments*⟩ have the following form:

`\p`⟨*point register*⟩`=`⟨*coordinate*⟩

Point position registers store a single point, consisting of an x-part and a y-part measured in TEX points (`pt`). In particular, point registers do not store nodes or node names. Here is an example:



```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw let \p{foo} = (1,1), \p2 = (2,0) in
          (0,0) -- (\p2) -- (\p{foo});
\end{tikzpicture}
```

`\p{`⟨*point register*⟩`}`

When this macro is used on the left-hand side of an `=`-sign in a let operation, it has no effect and is just there for readability. When the macro is used on the right-hand side of an `=`-sign or in the body of the let operation, then it expands to the x-part (measured in TEX points) of the coordinate stored in the ⟨*register*⟩, followed, by a comma, followed by the y-part.

For instance, if we say `let \p1=(1pt,1pt+2pt) in ...`, then inside the `...` part the macro `\p1` will expand to exactly the seven characters "1pt,3pt". This means that you when you write `(\p1)`, this expands to `(1pt,3pt)`, which is presumably exactly what you intended.

`\x{`⟨*point register*⟩`}`

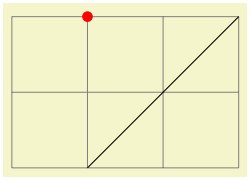
This macro expands just to the x-part of the point register. If we say as above, as we did above, `let \p1=(1pt,1pt+2pt) in ...`, then inside the `...` part the macro `\x1` expands to `1pt`.

`\y{`⟨*point register*⟩`}`

Works like `\x`, only for the y-part.

Note that the above macros are available only inside a let operation.

Here is an example where let clauses are used to assemble a coordinate from the x-coordinate of a first point and the y-coordinate of a second point. Naturally, using the `|-` notation, this could be written much more compactly.

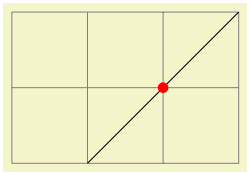


```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw    (1,0) coordinate (first point)
        -- (3,2) coordinate (second point);

  \fill[red] let \p1 = (first point),
                 \p2 = (second point) in
               (\x1,\y2) circle [radius=2pt];
\end{tikzpicture}
```

Note that the effect of a let operation is local to the body of the let operation. If you wish to access a computed coordinate outside the body, you must use a `coordinate` path operation:

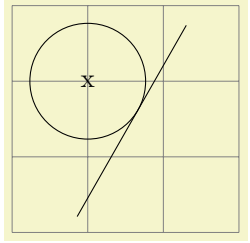


```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \path % let's define some points:
    let
      \p1        = (1,0),
      \p2        = (3,2),
      \p{center} = ($ (\p1) !.5! (\p2) $) % center
    in
      coordinate (p1) at (\p1)
      coordinate (p2) at (\p2)
      coordinate (center) at (\p{center});

  \draw (p1) -- (p2);
  \fill[red] (center) circle [radius=2pt];
\end{tikzpicture}
```

For a more useful application of the let operation, let us draw a circle that touches a given line:




```
\usetikzlibrary {calc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,3);

  \coordinate (a) at (rnd,rnd);
  \coordinate (b) at (3-rnd,3-rnd);
  \draw (a) -- (b);

  \node (c) at (1,2) {x};

  \draw let \p1 = ($ (a)!(c)!(b) - (c) $),
            \n1 = {veclen(\x1,\y1)}
        in circle [at=(c), radius=\n1];
\end{tikzpicture}
```

14.16 The Scoping Operation

When Ti*k*Z encounters and opening or a closing brace (`{` or `}`) at some point where a path operation should come, it will open or close a scope. All options that can be applied "locally" will be scoped inside the scope. For example, if you apply a transformation like `[xshift=1cm]` inside the scoped area, the shifting only applies to the scope. On the other hand, an option like `color=red` does not have any effect inside a scope since it can only be applied to the path as a whole.

Concerning the effect of scopes on relative coordinates, please see Section 13.4.3.

14.17 The Node and Edge Operations

The `node` operation adds a so-called node to a path. This operation is special in the following sense: It does not change the current path in any way. In other words, this operation is not really a path operation, but

has an effect that is "external" to the path. The `edge` operation has similar effect in that it adds something *after* the main path has been drawn. However, it works like the `to` operation, that is, it adds a `to` path to the picture after the main path has been drawn.

Since these operations are quite complex, they are described in the separate Section 17.

14.18 The Graph Operation

The `graph` operation can be used to specify easily how a large number of nodes are connected. This operation is documented in a separate section, see Section 19.

14.19 The Pic Operation

The `pic` operation is used to insert a "short picture" (hence the "short" name) at the current position of the path. This operation is somewhat similar to the `node` operation and discussed in detail in Section 18.

14.20 The Attribute Animation Operation

`\path` … `:`⟨*animation attribute*⟩`={`⟨*options*⟩`}` … `;`

This path operation has the same effect as if you had said:

```
[animate = { myself:⟨animate attribute⟩={⟨options⟩}} ]
```

This causes an animation of ⟨*animate attribute*⟩ to be added to the current path, see Section 26 for details.



```
\usetikzlibrary {animations}
\tikz \draw :xshift = {0s = "0cm", 30s = "-3cm", repeats} (0,0) circle [radius=5mm];
```

14.21 The PGF-Extra Operation

In some cases you may need to "do some calculations or some other stuff" while a path is constructed. For this, you would like to suspend the construction of the path and suspend Ti*k*Z's parsing of the path, you would then like to have some TEX code executed, and would then like to resume the parsing of the path. This effect can be achieved using the following path operation `\pgfextra`. Note that this operation should only be used by real experts and should only be used deep inside clever macros, not on normal paths. To add a tiny bit of safety when using it on a path anyway and the contents of `\pgfextra` contain drawing commands, those should be enclosed between `\pgfinterruptpath` and `\endpgfinterruptpath`.

`\pgfextra{`⟨*code*⟩`}`

This command may only be used inside a Ti*k*Z path. There it is used like a normal path operation. The construction of the path is temporarily suspended and the ⟨*code*⟩ is executed. Then, the path construction is resumed.

```
\newdimen\mydim
\begin{tikzpicture}
  \mydim=1cm
  \draw (0pt,\mydim) \pgfextra{\mydim=2cm} -- (0pt,\mydim);
\end{tikzpicture}
```

`\pgfextra`⟨*code*⟩ `\endpgfextra`

This is an alternative syntax for the `\pgfextra` command. If the code following `\pgfextra` does not start with a brace, the ⟨*code*⟩ is executed until `\endpgfextra` is encountered. What actually happens is that when `\pgfextra` is not followed by a brace, this completely shuts down the Ti*k*Z parser and `\endpgfextra` is a normal macro that restarts the parser.

```
\newdimen\mydim
\begin{tikzpicture}
  \mydim=1cm
  \draw (0pt,\mydim)
    \pgfextra \mydim=2cm \endpgfextra -- (0pt,\mydim);
\end{tikzpicture}
```

14.22 Interacting with the Soft Path subsystem

During construction Ti*k*Z stores the path internally as a *soft path*. Sometimes it is desirable to save a path during the stage of construction, restore it elsewhere and continue using it. There are two keys to facilitate this operation, which are explained below. To learn more about the soft path subsystem, refer to section 121.

`/tikz/save path=`⟨*macro*⟩ (no default)

Save the current soft path into ⟨*macro*⟩.

`/tikz/use path=`⟨*macro*⟩ (no default)

Set the current path to the soft path stored in ⟨*macro*⟩.



```
\usetikzlibrary {intersections}
\begin{tikzpicture}
  \path[save path=\pathA,name path=A] (0,1) to [bend left] (1,0);
  \path[save path=\pathB,name path=B]
    (0,0) .. controls (.33,.1) and (.66,.9) .. (1,1);

  \fill[name intersections={of=A and B}] (intersection-1) circle [radius=1pt];

  \draw[blue][use path=\pathA];
  \draw[red] [use path=\pathB];
\end{tikzpicture}
```

15 Actions on Paths

15.1 Overview

Once a path has been constructed, different things can be done with it. It can be drawn (or stroked) with a "pen", it can be filled with a color or shading, it can be used for clipping subsequent drawing, it can be used to specify the extend of the picture – or any combination of these actions at the same time.

To decide what is to be done with a path, two methods can be used. First, you can use a special-purpose command like `\draw` to indicate that the path should be drawn. However, commands like `\draw` and `\fill` are just abbreviations for special cases of the more general method: Here, the `\path` command is used to specify the path. Then, options encountered on the path indicate what should be done with the path.

For example, `\path (0,0) circle [radius=1cm];` means: "This is a path consisting of a circle around the origin. Do not do anything with it (throw it away)." However, if the option `draw` is encountered anywhere on the path, the circle will be drawn. "Anywhere" is any point on the path where an option can be given, which is everywhere where a path command like `circle [radius=1cm]` or `rectangle (1,1)` or even just `(0,0)` would also be allowed. Thus, the following commands all draw the same circle:

```
\path [draw] (0,0) circle [radius=1cm];
\path (0,0) [draw] circle [radius=1cm];
\path (0,0) circle [radius=1cm] [draw];
```

Finally, `\draw (0,0) circle [radius=1cm];` also draws a path, because `\draw` is an abbreviation for `\path [draw]` and thus the command expands to the first line of the above example.

Similarly, `\fill` is an abbreviation for `\path[fill]` and `\filldraw` is an abbreviation for the command `\path[fill,draw]`. Since options accumulate, the following commands all have the same effect:

```
\path [draw,fill]   (0,0) circle [radius=1cm];
\path [draw] [fill] (0,0) circle [radius=1cm];
\path [fill] (0,0) circle [radius=1cm] [draw];
\draw [fill] (0,0) circle [radius=1cm];
\fill (0,0) [draw] circle [radius=1cm];
\filldraw (0,0) circle [radius=1cm];
```

In the following subsection the different actions that can be performed on a path are explained. The following commands are abbreviations for certain sets of actions, but for many useful combinations there are no abbreviations:

`\draw`

Inside `{tikzpicture}` this is an abbreviation for `\path[draw]`.

`\fill`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill]`.

`\filldraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[fill,draw]`.

`\pattern`

Inside `{tikzpicture}` this is an abbreviation for `\path[pattern]`.

`\shade`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade]`.

`\shadedraw`

Inside `{tikzpicture}` this is an abbreviation for `\path[shade,draw]`.

`\clip`

Inside `{tikzpicture}` this is an abbreviation for `\path[clip]`.

`\useasboundingbox`

Inside `{tikzpicture}` this is an abbreviation for `\path[use as bounding box]`.

15.2 Specifying a Color

The most unspecific option for setting colors is the following:

`/tikz/color=`⟨*color name*⟩ (no default)

This option sets the color that is used for fill, drawing, and text inside the current scope. Any special settings for filling colors or drawing colors are immediately "overruled" by this option.

The ⟨*color name*⟩ is the name of a previously defined color. For LaTeX users, this is just a normal "LaTeX-color" and the `xcolor` extensions are allowed. Here is an example:

```
\tikz \fill[color=red!20] (0,0) circle [radius=1ex];
```

It is possible to "leave out" the `color=` part and you can also write:

```
\tikz \fill[red!20] (0,0) circle [radius=1ex];
```

What happens is that every option that Ti*k*Z does not know, like `red!20`, gets a "second chance" as a color name.

For plain TeX users, it is not so easy to specify colors since plain TeX has no "standardized" color naming mechanism. Because of this, PGF emulates the `xcolor` package, though the emulation is *extremely basic* (more precisely, what I could hack together in two hours or so). The emulation allows you to do the following:

- Specify a new color using `\definecolor`. Only the color models `gray`, `rgb`, and `RGB` are supported[3].

  Example: `\definecolor{orange}{rgb}{1,0.5,0}`
- Use `\colorlet` to define a new color based on an old one. Here, the `!` mechanism is supported, though only "once" (use multiple `\colorlet` for more fancy colors).

  Example: `\colorlet{lightgray}{black!25}`
- Use `\color{`⟨*color name*⟩`}` to set the color in the current TeX group. `\aftergroup`-hackery is used to restore the color after the group.

As pointed out above, the `color=` option applies to "everything" (except to shadings), which is not always what you want. Because of this, there are several more specialized color options. For example, the `draw=` option sets the color used for drawing, but does not modify the color used for filling. These color options are documented where the path action they influence is described.

15.3 Drawing a Path

You can draw a path using the following option:

`/tikz/draw=`⟨*color*⟩ (default is scope's color setting)

Causes the path to be drawn. "Drawing" (also known as "stroking") can be thought of as picking up a pen and moving it along the path, thereby leaving "ink" on the canvas.

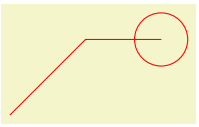
There are numerous parameters that influence how a line is drawn, like the thickness or the dash pattern. These options are explained below.

If the optional ⟨*color*⟩ argument is given, drawing is done using the given ⟨*color*⟩. This color can be different from the current filling color, which allows you to draw and fill a path with different colors. If no ⟨*color*⟩ argument is given, the last usage of the `color=` option is used.

If the special color name `none` is given, this option causes drawing to be "switched off". This is useful if a style has previously switched on drawing and you locally wish to undo this effect.

Although this option is normally used on paths to indicate that the path should be drawn, it also makes sense to use the option with a `{scope}` or `{tikzpicture}` environment. However, this will *not* cause all paths to be drawn. Instead, this just sets the ⟨*color*⟩ to be used for drawing paths inside the environment.

[3] ConTeXt users should be aware that `\definecolor` has a different meaning in ConTeXt. There is a low-level equivalent named `\pgfutil@definecolor` which can be used instead.



```
\begin{tikzpicture}
  \path[draw=red] (0,0) -- (1,1) -- (2,1) circle [radius=10pt];
\end{tikzpicture}
```

The following subsections list the different options that influence how a path is drawn. All of these options only have an effect if the `draw` option is given (directly or indirectly).

15.3.1 Graphic Parameters: Line Width, Line Cap, and Line Join

`/tikz/line width=`⟨*dimension*⟩ (no default, initially `0.4pt`)

Specifies the line width. Note the space.



```
\tikz \draw[line width=5pt] (0,0) -- (1cm,1.5ex);
```

There are a number of predefined styles that provide more "natural" ways of setting the line width. You can also redefine these styles.

`/tikz/ultra thin` (style, no value)

Sets the line width to 0.1pt.



```
\tikz \draw[ultra thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/very thin` (style, no value)

Sets the line width to 0.2pt.



```
\tikz \draw[very thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/thin` (style, no value)

Sets the line width to 0.4pt.



```
\tikz \draw[thin] (0,0) -- (1cm,1.5ex);
```

`/tikz/semithick` (style, no value)

Sets the line width to 0.6pt.



```
\tikz \draw[semithick] (0,0) -- (1cm,1.5ex);
```

`/tikz/thick` (style, no value)

Sets the line width to 0.8pt.



```
\tikz \draw[thick] (0,0) -- (1cm,1.5ex);
```

`/tikz/very thick` (style, no value)

Sets the line width to 1.2pt.



```
\tikz \draw[very thick] (0,0) -- (1cm,1.5ex);
```

/tikz/ultra thick (style, no value)

Sets the line width to 1.6pt.



```
\tikz \draw[ultra thick] (0,0) -- (1cm,1.5ex);
```

/tikz/line cap=⟨*type*⟩ (no default, initially `butt`)

Specifies how lines "end". Permissible ⟨*type*⟩ are `round`, `rect`, and `butt`. They have the following effects:



```
\begin{tikzpicture}
  \begin{scope}[line width=10pt]
    \draw[line cap=round] (0,1 ) -- +(1,0);
    \draw[line cap=butt]  (0,.5) -- +(1,0);
    \draw[line cap=rect]  (0,0 ) -- +(1,0);
  \end{scope}
  \draw[white,line width=1pt]
    (0,0 ) -- +(1,0) (0,.5) -- +(1,0) (0,1 ) -- +(1,0);
\end{tikzpicture}
```

/tikz/line join=⟨*type*⟩ (no default, initially `miter`)

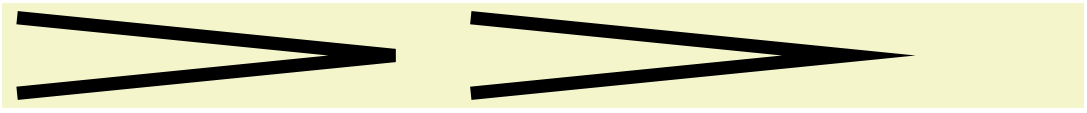
Specifies how lines "join". Permissible ⟨*type*⟩ are `round`, `bevel`, and `miter`. They have the following effects:



```
\begin{tikzpicture}[line width=10pt]
  \draw[line join=round] (0,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=bevel] (1.25,0) -- ++(.5,1) -- ++(.5,-1);
  \draw[line join=miter] (2.5,0) -- ++(.5,1) -- ++(.5,-1);
  \useasboundingbox (0,1.5); % enlarge bounding box
\end{tikzpicture}
```

/tikz/miter limit=⟨*factor*⟩ (no default, initially `10`)

When you use the miter join and there is a very sharp corner (a small angle), the miter join may protrude very far over the actual joining point. In this case, if it were to protrude by more than ⟨*factor*⟩ times the line width, the miter join is replaced by a bevel join.



```
\begin{tikzpicture}[line width=5pt]
  \draw                 (0,0) -- ++(5,.5) -- ++(-5,.5);
  \draw[miter limit=25] (6,0) -- ++(5,.5) -- ++(-5,.5);
  \useasboundingbox (14,0); % make bounding box bigger
\end{tikzpicture}
```

15.3.2 Graphic Parameters: Dash Pattern

/tikz/dash pattern=⟨*dash pattern*⟩ (no default)

Sets the dashing pattern. The syntax is the same as in METAFONT. For example following pattern `on 2pt off 3pt on 4pt off 4pt` means "draw 2pt, then leave out 3pt, then draw 4pt once more, then leave out 4pt again, repeat".


```
\begin{tikzpicture}[dash pattern=on 2pt off 3pt on 4pt off 4pt]
  \draw (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

/tikz/dash phase=⟨*dash phase*⟩ (no default, initially `0pt`)

Shifts the start of the dash pattern by ⟨*phase*⟩.


```
\begin{tikzpicture}[dash pattern=on 20pt off 10pt]
  \draw[dash phase=0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw[dash phase=10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

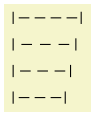
/tikz/dash=⟨*dash pattern*⟩`phase`⟨*dash phase*⟩ (no default)

Sets the dashing pattern and phase at the same time.


```
\begin{tikzpicture}
  \draw [dash=on 20pt off 10pt phase  0pt] (0pt,3pt) -- (3.5cm,3pt);
  \draw [dash=on 20pt off 10pt phase 10pt] (0pt,0pt) -- (3.5cm,0pt);
\end{tikzpicture}
```

/tikz/dash expand off (no value)

Makes the `off` part of a dash pattern expandable such that it can stretch. This only works when there is a single `on` and a single `off` field and requires the `decorations` library. Right now this option has to be specified on the path where it is supposed to take effect after the `dash pattern` option because the dash pattern has to be known at the point where it is applied.



```
\usetikzlibrary {decorations}
\begin{tikzpicture}[|-|, dash pattern=on 4pt off 2pt]
  \draw [dash expand off] (0pt,30pt) -- (26pt,30pt);
  \draw [dash expand off] (0pt,20pt) -- (24pt,20pt);
  \draw [dash expand off] (0pt,10pt) -- (22pt,10pt);
  \draw [dash expand off] (0pt, 0pt) -- (20pt, 0pt);
\end{tikzpicture}
```

As for the line thickness, some predefined styles allow you to set the dashing conveniently.

/tikz/solid (style, no value)

Shorthand for setting a solid line as “dash pattern”. This is the default.


```
\tikz \draw[solid] (0pt,0pt) -- (50pt,0pt);
```

/tikz/dotted (style, no value)

Shorthand for setting a dotted dash pattern.


```
\tikz \draw[dotted] (0pt,0pt) -- (50pt,0pt);
```

/tikz/densely dotted (style, no value)

Shorthand for setting a densely dotted dash pattern.


```
\tikz \draw[densely dotted] (0pt,0pt) -- (50pt,0pt);
```

/tikz/loosely dotted (style, no value)

Shorthand for setting a loosely dotted dash pattern.


```
\tikz \draw[loosely dotted] (0pt,0pt) -- (50pt,0pt);
```

/tikz/dashed (style, no value)

Shorthand for setting a dashed dash pattern.


```
\tikz \draw[dashed] (0pt,0pt) -- (50pt,0pt);
```

/tikz/densely dashed (style, no value)

Shorthand for setting a densely dashed dash pattern.


```
\tikz \draw[densely dashed] (0pt,0pt) -- (50pt,0pt);
```

/tikz/loosely dashed (style, no value)

Shorthand for setting a loosely dashed dash pattern.


```
\tikz \draw[loosely dashed] (0pt,0pt) -- (50pt,0pt);
```

/tikz/dash dot (style, no value)

Shorthand for setting a dashed and dotted dash pattern.


```
\tikz \draw[dash dot] (0pt,0pt) -- (50pt,0pt);
```

/tikz/densely dash dot (style, no value)

Shorthand for setting a densely dashed and dotted dash pattern.


```
\tikz \draw[densely dash dot] (0pt,0pt) -- (50pt,0pt);
```

/tikz/loosely dash dot (style, no value)

Shorthand for setting a loosely dashed and dotted dash pattern.


```
\tikz \draw[loosely dash dot] (0pt,0pt) -- (50pt,0pt);
```

/tikz/dash dot dot (style, no value)

Shorthand for setting a dashed and dotted dash pattern with more dots.


```
\tikz \draw[dash dot dot] (0pt,0pt) -- (50pt,0pt);
```

/tikz/densely dash dot dot (style, no value)

Shorthand for setting a densely dashed and dotted dash pattern with more dots.


```
\tikz \draw[densely dash dot dot] (0pt,0pt) -- (50pt,0pt);
```

/tikz/loosely dash dot dot (style, no value)

Shorthand for setting a loosely dashed and dotted dash pattern with more dots.


```
\tikz \draw[loosely dash dot dot] (0pt,0pt) -- (50pt,0pt);
```

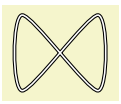
15.3.3 Graphic Parameters: Draw Opacity

When a line is drawn, it will normally "obscure" everything behind it as if you had used perfectly opaque ink. It is also possible to ask Ti*k*Z to use an ink that is a little bit (or a big bit) transparent using the `draw opacity` option. This is explained in Section 23 on transparency in more detail.

15.3.4 Graphic Parameters: Double Lines and Bordered Lines

/tikz/double=⟨*core color*⟩ (default white)

This option causes "two" lines to be drawn instead of a single one. However, this is not what really happens. In reality, the path is drawn twice. First, with the normal drawing color, secondly with the ⟨*core color*⟩, which is normally white. Upon the second drawing, the line width is reduced. The net effect is that it appears as if two lines had been drawn and this works well even with complicated, curved paths:



```
\tikz \draw[double]
  plot[smooth cycle] coordinates{(0,0) (1,1) (1,0) (0,1)};
```

You can also use the doubling option to create an effect in which a line seems to have a certain "border":



```
\begin{tikzpicture}
  \draw (0,0) -- (1,1);
  \draw[draw=white,double=red,very thick] (0,1) -- (1,0);
\end{tikzpicture}
```

/tikz/double distance=⟨*dimension*⟩ (no default, initially 0.6pt)

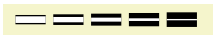
Sets the distance the "two" lines are spaced apart. In reality, this is the thickness of the line that is used to draw the path for the second time. The thickness of the *first* time the path is drawn is twice the normal line width plus the given ⟨*dimension*⟩. As a side-effect, this option "selects" the double option.



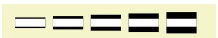
```
\begin{tikzpicture}
  \draw[very thick,double]              (0,0) arc (180:90:1cm);
  \draw[very thick,double distance=2pt] (1,0) arc (180:90:1cm);
  \draw[thin,double distance=2pt]       (2,0) arc (180:90:1cm);
\end{tikzpicture}
```

/tikz/double distance between line centers=⟨*dimension*⟩ (no default)

This option works like double distance, only the distance is not the distance between (inner) borders of the two main lines, but between their centers. Thus, the thickness the *first* time the path is drawn is the normal line width plus the given ⟨*dimension*⟩, while the line width of the *second* line that is drawn is ⟨*dimension*⟩ minus the normal line width. As a side-effect, this option "selects" the double option.



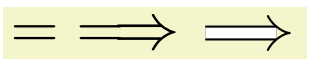
```
\begin{tikzpicture}[double distance between line centers=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```



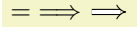
```
\begin{tikzpicture}[double distance=3pt]
  \foreach \lw in {0.5,1,1.5,2,2.5}
    \draw[line width=\lw pt,double] (\lw,0) -- ++(4mm,0);
\end{tikzpicture}
```

/tikz/double equal sign distance (style, no value)

This style selects a double line distance such that it corresponds to the distance of the two lines in an equal sign.



```
\usepackage {amsmath} \usetikzlibrary {arrows.meta}
\Huge $=\implies$\tikz[baseline,double equal sign distance]
                    \draw[double,thick,-{Implies[]}](0,0.55ex) --++(3ex,0);
```



```
\usepackage {amsmath} \usetikzlibrary {arrows.meta}
\normalsize $=\implies$\tikz[baseline,double equal sign distance]
                          \draw[double,-{Implies[]}](0,0.6ex) --++(3ex,0);
```

```
\usepackage {amsmath} \usetikzlibrary {arrows.meta}
\tiny $=\implies$\tikz[baseline,double equal sign distance]
                    \draw[double,very thin,-{Implies[]}](0,0.5ex) -- ++(3ex,0);
```

15.4 Adding Arrow Tips to a Path

In different situations, TikZ will add arrow tips to the end of a path. For this to happen, a number of different things need to be specified:

1. You must have used the `arrows` key, explained in detail in Section 16, to setup which kinds of arrow tips you would like.
2. The path may not be closed (like a circle or a rectangle) and, if it consists of several subpaths, further restrictions apply as explained in Section 16.
3. The `tips` key must be set to an appropriate value, see Section 16 once more.

For the current section on paths, it is only important that when you add the `tips` option to a path that is not drawn, arrow tips will still be added at the beginning and at the end of the current path. This is true even when "only" arrow tips get drawn for a path without drawing the path itself. Here is an example:



```
\usetikzlibrary {arrows.meta,bending}
\tikz \path[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```



```
\usetikzlibrary {arrows.meta,bending}
\tikz \draw[tips, -{Latex[open,length=10pt,bend]}] (0,0) to[bend left] (1,0);
```

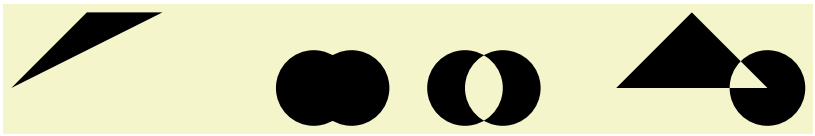
15.5 Filling a Path

To fill a path, use the following option:

`/tikz/fill=`⟨*color*⟩ (default is scope's color setting)

This option causes the path to be filled. All unclosed parts of the path are first closed, if necessary. Then, the area enclosed by the path is filled with the current filling color, which is either the last color set using the general `color=` option or the optional color ⟨*color*⟩. For self-intersection paths and for paths consisting of several closed areas, the "enclosed area" is somewhat complicated to define and two different definitions exist, namely the nonzero winding number rule and the even odd rule, see the explanation of these options, below.

Just as for the `draw` option, setting ⟨*color*⟩ to `none` disables filling locally.



```
\begin{tikzpicture}
  \fill (0,0) -- (1,1) -- (2,1);
  \fill (4,0) circle [radius=.5cm]  (4.5,0) circle [radius=.5cm];
  \fill[even odd rule] (6,0) circle [radius=.5cm]  (6.5,0) circle [radius=.5cm];
  \fill (8,0) -- (9,1) -- (10,0) circle [radius=.5cm];
\end{tikzpicture}
```

If the `fill` option is used together with the `draw` option (either because both are given as options or because a `\filldraw` command is used), the path is filled *first*, then the path is drawn *second*. This is especially useful if different colors are selected for drawing and for filling. Even if the same color is used, there is a difference between this command and a plain `fill`: A "filldrawn" area will be slightly larger than a filled area because of the thickness of the "pen".



```
\begin{tikzpicture}[fill=yellow!80!black,line width=5pt]
  \filldraw (0,0) -- (1,1) -- (2,1);
  \filldraw (4,0) circle [radius=.5cm]  (4.5,0) circle [radius=.5cm];
  \filldraw[even odd rule] (6,0) circle [radius=.5cm]  (6.5,0) circle [radius=.5cm];
  \filldraw (8,0) -- (9,1) -- (10,0) circle [radius=.5cm];
\end{tikzpicture}
```

15.5.1 Graphic Parameters: Fill Pattern

Instead of filling a path with a single solid color, it is also possible to fill it with a *tiling pattern*. Imagine a small tile that contains a simple picture like a star. Then these tiles are (conceptually) repeated infinitely in all directions, but clipped against the path.

Tiling patterns come in two variants: *inherently colored patterns* and *form-only patterns*. An inherently colored pattern is, say, a red star with a black border and will always look like this. A form-only pattern may have a different color each time it is used, only the form of the pattern will stay the same. As such, form-only patterns do not have any colors of their own, but when it is used the current *pattern color* is used as its color.

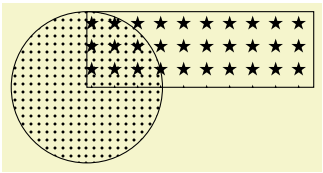
Patterns are not overly flexible. In particular, it is not possible to change the size or orientation of a pattern without declaring a new pattern. For complicated cases, it may be easier to use two nested `\foreach` statements to simulate a pattern, but patterns are rendered *much* more quickly than simulated ones.

`/tikz/pattern=`⟨*name*⟩ (default is scope's pattern)

This option causes the path to be filled with a pattern. If the ⟨*name*⟩ is given, this pattern is used, otherwise the pattern set in the enclosing scope is used. As for the `draw` and `fill` options, setting ⟨*name*⟩ to `none` disables filling locally.

The pattern works like a fill color. In particular, setting a new fill color will fill the path with a solid color once more.

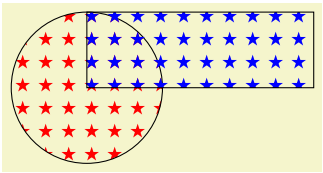
Strangely, no ⟨*name*⟩s are permissible by default. You need to load for instance the `patterns` library, see Section 62, to install predefined patterns.



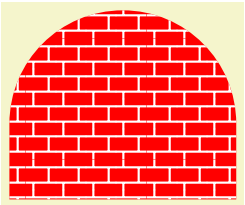
```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \draw[pattern=dots] (0,0) circle [radius=1cm];
  \draw[pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```

`/tikz/pattern color=`⟨*color*⟩ (no default)

This option is used to set the color to be used for form-only patterns. This option has no effect on inherently colored patterns.



```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \draw[pattern color=red,pattern=fivepointed stars]  (0,0) circle [radius=1cm];
  \draw[pattern color=blue,pattern=fivepointed stars] (0,0) rectangle (3,1);
\end{tikzpicture}
```



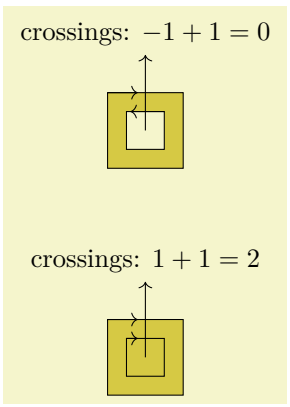
```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \def\mypath{(0,0) -- +(0,1) arc (180:0:1.5cm) -- +(0,-1)}
  \fill   [red]                                  \mypath;
  \pattern[pattern color=white,pattern=bricks] \mypath;
\end{tikzpicture}
```

15.5.2 Graphic Parameters: Interior Rules

The following two options can be used to decide how interior points should be determined:

`/tikz/nonzero rule` (no value)

If this rule is used (which is the default), the following method is used to determine whether a given point is "inside" the path: From the point, shoot a ray in some direction towards infinity (the direction is chosen such that no strange borderline cases occur). Then the ray may hit the path. Whenever it hits the path, we increase or decrease a counter, which is initially zero. If the ray hits the path as the path goes "from left to right" (relative to the ray), the counter is increased, otherwise it is decreased. Then, at the end, we check whether the counter is nonzero (hence the name). If so, the point is deemed to lie "inside", otherwise it is "outside". Sounds complicated? It is.




```
\begin{tikzpicture}
  \filldraw[fill=yellow!80!black]
  % Clockwise rectangle
  (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
  % Counter-clockwise rectangle
  (0.25,0.25) -- (0.75,0.25) -- (0.75,0.75) -- (0.25,0.75) -- cycle;

  \draw[->] (0,1) -- (.4,1);
  \draw[->] (0.75,0.75) -- (0.3,.75);

  \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $-1+1 = 0$};

  \begin{scope}[yshift=-3cm]
    \filldraw[fill=yellow!80!black]
    % Clockwise rectangle
    (0,0) -- (0,1) -- (1,1) -- (1,0) -- cycle
    % Clockwise rectangle
    (0.25,0.25) -- (0.25,0.75) -- (0.75,0.75) -- (0.75,0.25) -- cycle;

    \draw[->] (0,1) -- (.4,1);
    \draw[->] (0.25,0.75) -- (0.4,.75);

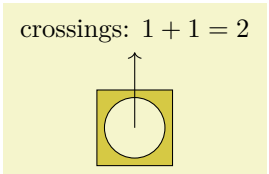
    \draw[->] (0.5,0.5) -- +(0,1) node[above] {crossings: $1+1 = 2$};
  \end{scope}
\end{tikzpicture}
```

`/tikz/even odd rule` (no value)

This option causes a different method to be used for determining the inside and outside of paths. While it is less flexible, it turns out to be more intuitive.

With this method, we also shoot rays from the point for which we wish to determine whether it is inside or outside the filling area. However, this time we only count how often we "hit" the path and declare the point to be "inside" if the number of hits is odd.

Using the even-odd rule, it is easy to "drill holes" into a path.




```
\begin{tikzpicture}
  \filldraw[fill=yellow!80!black,even odd rule]
    (0,0) rectangle (1,1) (0.5,0.5) circle [radius=0.4cm];
  \draw[->] (0.5,0.5) -- +(0,1) [above] node{crossings: $1+1 = 2$};
\end{tikzpicture}
```

15.5.3 Graphic Parameters: Fill Opacity

Analogously to the `draw opacity`, you can also set the fill opacity. Please see Section 23 for more details.

15.6 Generalized Filling: Using Arbitrary Pictures to Fill a Path

Sometimes you wish to "fill" a path with something even more complicated than a pattern, let alone a single color. For instance, you might wish to use an image to fill the path or some other, complicated drawing. In principle, this effect can be achieved by first using the path for clipping and then, subsequently, drawing the desired image or picture. However, there is an option that makes this process much easier:

`/tikz/path picture=`⟨*code*⟩ (no default)

When this option is given on a path and when the ⟨*code*⟩ is not empty, the following happens: After all other "filling" operations are done with the path, which are caused by the options `fill`, `pattern` and `shade`, a local scope is opened and the path is temporarily installed as a clipping path. Then, the ⟨*code*⟩ is executed, which can now draw something. Then, the local scope ends and, possibly, the path is stroked, provided the `draw` option has been given.

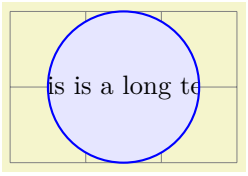
As with other keys like `fill` or `draw` this option needs to be given on a path, setting the `path picture` outside a path has no effect (the path picture is cleared at the beginning of each path).

The ⟨*code*⟩ can be any normal Ti*k*Z code like `\draw ...` or `\node ...`. As always, when you include an external graphic, you need to put it inside a `\node`.

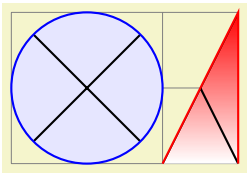
Note that no special actions are taken to transform the origin in any way. This means that the coordinate `(0,0)` is still where is was when the path was being constructed and not – as one might expect – at the lower left corner of the path. However, you can use the following special node to access the size of the path:

Predefined node `path picture bounding box`

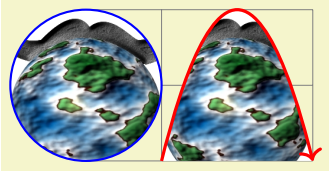
This node is of shape `rectangle`. Its size and position are those of `current path bounding box` just before the ⟨*code*⟩ of the path picture started to be executed. The ⟨*code*⟩ can construct its own paths, so accessing the `current path bounding box` inside the ⟨*code*⟩ yields the bounding box of any path that is currently being constructed inside the ⟨*code*⟩.




```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [fill=blue!10,draw=blue,thick] (1.5,1) circle [radius=1]
    [path picture={
      \node at (path picture bounding box.center) {
        This is a long text.
      };}
    ];
\end{tikzpicture}
```



```
\begin{tikzpicture}[cross/.style={path picture={
      \draw[black]
            (path picture bounding box.south east) --
            (path picture bounding box.north west)
            (path picture bounding box.south west) --
            (path picture bounding box.north east);
    }}]
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [cross,fill=blue!10,draw=blue,thick] (1,1) circle [radius=1];
  \path     [cross,top color=red,draw=red,thick] (2,0) -- (3,2) -- (3,0);
\end{tikzpicture}
```



```
\begin{tikzpicture}[path image/.style={
    path picture={
      \node at (path picture bounding box.center) {
        \includegraphics[height=3cm]{#1}
      };}}]
\draw     [help lines] (0,0) grid (3,2);

\draw [path image=brave-gnu-world-logo,draw=blue,thick]
        (0,1) circle [radius=1];
\draw [path image=brave-gnu-world-logo,draw=red,very thick,->]
        (1,0) parabola[parabola height=2cm] (3,0);

\end{tikzpicture}
```

15.7 Shading a Path

You can shade a path using the `shade` option. A shading is like a filling, only the shading changes its color smoothly from one color to another.

`/tikz/shade` (no value)

Causes the path to be shaded using the currently selected shading (more on this later). If this option is used together with the `draw` option, then the path is first shaded, then drawn.

It is not an error to use this option together with the `fill` option, but it makes no sense.


```
\tikz \shade (0,0) circle [radius=1ex];
```



```
\tikz \shadedraw (0,0) circle [radius=1ex];
```

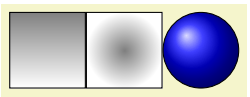
For some shadings it is not really clear how they can "fill" the path. For example, the `ball` shading normally looks like this: . How is this supposed to shade a rectangle? Or a triangle?

To solve this problem, the predefined shadings like `ball` or `axis` fill a large rectangle completely in a sensible way. Then, when the shading is used to "shade" a path, what actually happens is that the path is temporarily used for clipping and then the rectangular shading is drawn, scaled and shifted such that all parts of the path are filled.

The default shading is a smooth transition from gray to white and from top to bottom. However, other shadings are also possible, for example a shading that will sweep a color from the center to the corners outward. To choose the shading, you can use the `shading=` option, which will also automatically invoke the `shade` option. Note that this does *not* change the shading color, only the way the colors sweep. For changing the colors, other options are needed, which are explained below.

`/tikz/shading=`⟨*name*⟩ (no default)

This selects a shading named ⟨*name*⟩. The following shadings are predefined: `axis`, `radial`, and `ball`.



```
\tikz \shadedraw [shading=axis] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=radial] (0,0) rectangle (1,1);
\tikz \shadedraw [shading=ball] (0,0) circle [radius=.5cm];
```

The shadings as well as additional shadings are described in more detail in Section 69.

To change the color of a shading, special options are needed like `left color`, which sets the color of an axis shading from left to right. These options implicitly also select the correct shading type, see the following example



```
\tikz \shadedraw [left color=red,right color=blue]
    (0,0) rectangle (1,1);
```

For a complete list of the possible options see Section 69 once more.

`/tikz/shading angle=`⟨*degrees*⟩ (no default, initially `0`)

This option rotates the shading (not the path!) by the given angle. For example, we can turn a top-to-bottom axis shading into a left-to-right shading by rotating it by 90°.



```
\tikz \shadedraw [shading=axis,shading angle=90] (0,0) rectangle (1,1);
```

You can also define new shading types yourself. However, for this, you need to use the basic layer directly, which is, well, more basic and harder to use. Details on how to create a shading appropriate for filling paths are given in Section 114.3.

15.8 Establishing a Bounding Box

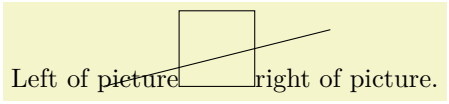
PGF is reasonably good at keeping track of the size of your picture and reserving just the right amount of space for it in the main document. However, in some cases you may want to say things like "do not count this for the picture size" or "the picture is actually a little large". For this you can use the option `use as bounding box` or the command `\useasboundingbox`, which is just a shorthand for `\path[use as bounding box]`.

`/tikz/use as bounding box` (no value)

Normally, when this option is given on a path, the bounding box of the present path is used to determine the size of the picture and the size of all *subsequent* paths are ignored. However, if there were previous path operations that have already established a larger bounding box, it will not be made smaller by this operation (consider the `\pgfresetboundingbox` command to reset the previous bounding box).

In a sense, `use as bounding box` has the same effect as clipping all subsequent drawing against the current path – without actually doing the clipping, only making PGF treat everything as if it were clipped.

The first application of this option is to have a `{tikzpicture}` overlap with the main text:




```
Left of picture\begin{tikzpicture}
  \draw[use as bounding box] (2,0) rectangle (3,1);
  \draw (1,0) -- (4,.75);
\end{tikzpicture}right of picture.
```

In a second application this option can be used to get better control over the white space around the picture:



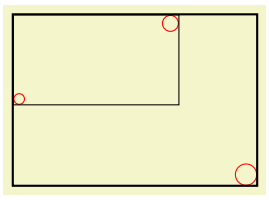

```
Left of picture
\begin{tikzpicture}
  \useasboundingbox (0,0) rectangle (3,1);
  \fill (.75,.25) circle [radius=.5cm];
\end{tikzpicture}
right of picture.
```

Note: If this option is used on a path inside a TEX group (scope), the effect "lasts" only until the end of the scope. Again, this behavior is the same as for clipping.

Consider using `\useasboundingbox` together with `\pgfresetboundingbox` in order to replace the bounding box with a new one.

There is a node that allows you to get the size of the current bounding box. The `current bounding box` node has the `rectangle` shape and its size is always the size of the current bounding box.

Similarly, the `current path bounding box` node has the `rectangle` shape and the size of the bounding box of the current path.



```
\begin{tikzpicture}
  \draw[red] (0,0) circle [radius=2pt];
  \draw[red] (2,1) circle [radius=3pt];

  \draw (current bounding box.south west) rectangle
        (current bounding box.north east);

  \draw[red] (3,-1) circle [radius=4pt];

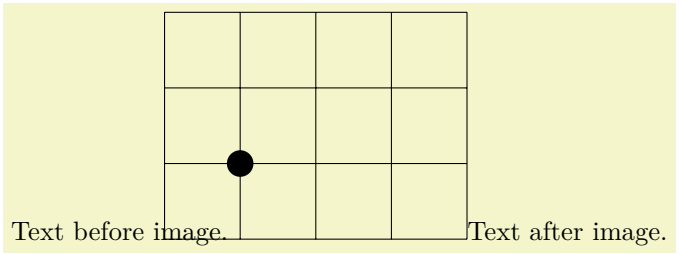
  \draw[thick] (current bounding box.south west) rectangle
               (current bounding box.north east);
\end{tikzpicture}
```

Occasionally, you may want to align multiple `tikzpicture` environments horizontally and/or vertically at some prescribed position. The vertical alignment can be realized by means of the `baseline` option since

TEX supports the concept of box depth natively. For horizontal alignment, things are slightly more involved. The following approach is realized by means of negative `\hspace`s before and/or after the picture, thereby removing parts of the picture. However, the actual amount of negative horizontal space is provided by means of image coordinates using the `trim left` and `trim right` keys:

`/tikz/trim left=`⟨*dimension or coordinate or* ***default***⟩ (default `0pt`)

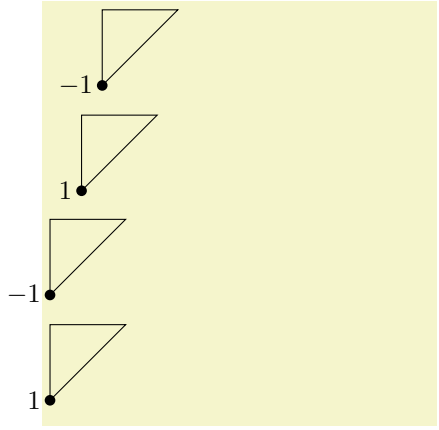
The `trim left` key tells PGF to discard everything which is left of the provided ⟨*dimension or coordinate*⟩. Here, ⟨*dimension*⟩ is a single x coordinate of the picture and ⟨*coordinate*⟩ is a point with x and y coordinates (but only its x coordinate will be used). The effect is the same as if you issue `\hspace{-s}` where `s` is the difference of the picture's bounding box lower left x coordinate and the x coordinate specified as ⟨*dimension or coordinate*⟩:




```
Text before image.%
    \begin{tikzpicture}[trim left]
        \draw (-1,-1) grid (3,2);
        \fill (0,0) circle [radius=5pt];
    \end{tikzpicture}%
Text after image.
```

Since `trim left` uses the default `trim left=0pt`, everything left of $x = 0$ is removed from the bounding box.

The following example has once the relative long label -1 and once the shorter label 1. Horizontal alignment is established with `trim left`:



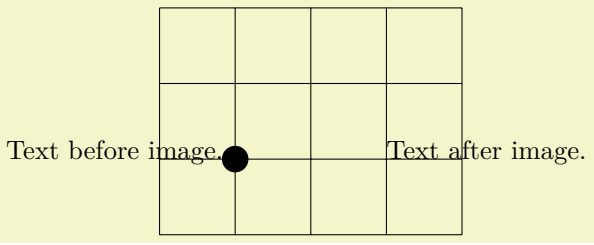

```
\begin{tikzpicture}
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle [radius=2pt];
    \node[left] at (0,0) {$-1$};
\end{tikzpicture}
\par
\begin{tikzpicture}
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle [radius=2pt];
    \node[left] at (0,0) {$1$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle [radius=2pt];
    \node[left] at (0,0) {$-1$};
\end{tikzpicture}
\par
\begin{tikzpicture}[trim left]
    \draw (0,1) -- (0,0) -- (1,1) -- cycle;
    \fill (0,0) circle [radius=2pt];
    \node[left] at (0,0) {$1$};
\end{tikzpicture}
```

Use `trim left=default` to reset the value.

`/tikz/trim right=`⟨*dimension or coordinate or* ***default***⟩ (no default)

This key is similar to `trim left`: it discards everything which is right of the provided ⟨*dimension or coordinate*⟩. As for `trim left`, ⟨*dimension*⟩ denotes a single x coordinate of the picture and ⟨*coordinate*⟩ a coordinate with x and y value (although only its x component will be used).

We use the same example from above and add `trim right`:




```
Text before image.%
    \begin{tikzpicture}[trim left, trim right=2cm, baseline]
        \draw (-1,-1) grid (3,2);
        \fill (0,0) circle [radius=5pt];
    \end{tikzpicture}%
Text after image.
```

In addition to `trim left=0pt`, we also discard everything which is right of x=`2cm`. Furthermore, the `baseline` key supports vertical alignment as well (using the y=`0cm` baseline).

Use `trim right=default` to reset the value.

Note that `baseline`, `trim left` and `trim right` are currently the *only* supported way of truncated bounding boxes which are compatible with image externalization (see the `external` library for details).

`/pgf/trim lowlevel=true|false` (no default, initially `false`)

This affects only the basic level image externalization: the initial configuration `trim lowlevel=false` stores the normal image, without trimming, and the trimming into a separate file. This allows reduced bounding boxes without clipping the rest away. The `trim lowlevel=true` information causes the image externalization to store the trimmed image, possibly resulting in clipping.

15.9 Clipping and Fading (Soft Clipping)

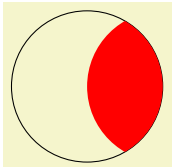
Clipping path means that all painting on the page is restricted to a certain area. This area need not be rectangular, rather an arbitrary path can be used to specify this area. The `clip` option, explained below, is used to specify the region that is to be used for clipping.

A *fading* (a term that I propose, fadings are commonly known as soft masks, transparency masks, opacity masks or soft clips) is similar to clipping, but a fading allows parts of the picture to be only "half clipped". This means that a fading can specify that newly painted pixels should be partly transparent. The specification and handling of fadings is a bit complex and it is detailed in Section 23, which is devoted to transparency in general.

`/tikz/clip` (no value)

This option causes all subsequent drawings to be clipped against the current path and the size of subsequent paths will not be important for the picture size. If you clip against a self-intersecting path, the even-odd rule or the nonzero winding number rule is used to determine whether a point is inside or outside the clipping region.

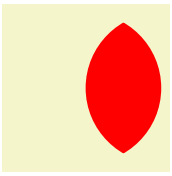
The clipping path is a graphic state parameter, so it will be reset at the end of the current scope. Multiple clippings accumulate, that is, clipping is always done against the intersection of all clipping areas that have been specified inside the current scopes. The only way of enlarging the clipping area is to end a `{scope}`.



```
\begin{tikzpicture}
  \draw[clip] (0,0) circle [radius=1cm];
  \fill[red] (1,0) circle [radius=1cm];
\end{tikzpicture}
```

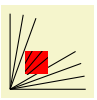
It is usually a *very* good idea to apply the `clip` option only to the first path command in a scope.

If you "only wish to clip" and do not wish to draw anything, you can use the `\clip` command, which is a shorthand for `\path[clip]`.



```
\begin{tikzpicture}
  \clip (0,0) circle [radius=1cm];
  \fill[red] (1,0) circle [radius=1cm];
\end{tikzpicture}
```

To keep clipping local, use `{scope}` environments as in the following example:



```
\begin{tikzpicture}
  \draw (0,0) -- ( 0:1cm);
  \draw (0,0) -- (10:1cm);
  \draw (0,0) -- (20:1cm);
  \draw (0,0) -- (30:1cm);
  \begin{scope}[fill=red]
    \fill[clip] (0.2,0.2) rectangle (0.5,0.5);

    \draw (0,0) -- (40:1cm);
    \draw (0,0) -- (50:1cm);
    \draw (0,0) -- (60:1cm);
  \end{scope}
  \draw (0,0) -- (70:1cm);
  \draw (0,0) -- (80:1cm);
  \draw (0,0) -- (90:1cm);
\end{tikzpicture}
```

There is a slightly annoying catch: You cannot specify certain graphic options for the command used for clipping. For example, in the above code we could not have moved the `fill=red` to the `\fill` command. The reasons for this have to do with the internals of the PDF specification. You do not want to know the details. It is best simply not to specify any options for these commands.

15.10 Doing Multiple Actions on a Path

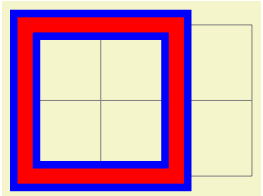
If more than one of the basic actions like drawing, clipping and filling are requested, they are automatically applied in a sensible order: First, a path is filled, then drawn, and then clipped (although it took Apple two major revisions of their operating system to get this right…). Sometimes, however, you need finer control over what is done with a path. For instance, you might wish to first fill a path with a color, then repaint the path with a pattern and then repaint it with yet another pattern. In such cases you can use the following two options:

`/tikz/preaction`=⟨*options*⟩ (no default)

This option can be given to a `\path` command (or to derived commands like `\draw` which internally call `\path`). Similarly to options like `draw`, this option only has an effect when given to a `\path` or as part of the options of a `node`; as an option to a `{scope}` it has no effect.

When this option is used on a `\path`, the effect is the following: When the path has been completely constructed and is about to be used, a scope is created. Inside this scope, the path is used but not with the original path options, but with ⟨*options*⟩ instead. Then, the path is used in the usual manner. In other words, the path is used twice: Once with ⟨*options*⟩ in force and then again with the normal path options in force.

Here is an example in which the path consists of a rectangle. The main action is to draw this path in red (which is why we see a red rectangle). However, the preaction is to draw the path in blue, which is why we see a blue rectangle behind the red rectangle.

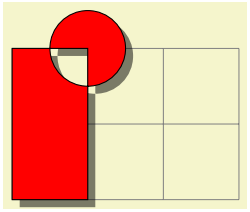


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \draw
    [preaction={draw,line width=4mm,blue}]
    [line width=2mm,red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

Note that when the preactions are performed, then the path is already "finished". In particular, applying a coordinate transformation to the path has no effect. By comparison, applying a canvas transformation

does have an effect. Let us use this to add a "shadow" to a path. For this, we use the preaction to fill the path in gray, shifted a bit to the right and down:

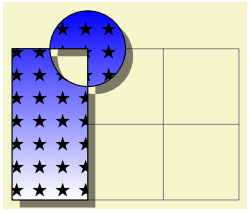


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw
    [preaction={fill=black,opacity=.5,
                transform canvas={xshift=1mm,yshift=-1mm}}]
    [fill=red] (0,0) rectangle (1,2)
               (1,2) circle [radius=5mm];
\end{tikzpicture}
```

Naturally, you would normally create a style `shadow` that contains the above code. The `shadows` library, see Section 70, contains predefined shadows of this kind.

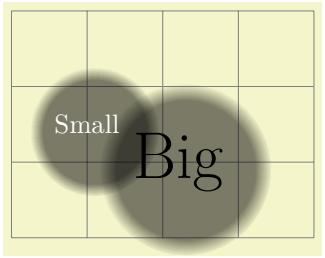
It is possible to use the `preaction` option multiple times. In this case, for each use of the `preaction` option, the path is used again (thus, the ⟨*options*⟩ do not accumulate in a single usage of the path). The path is used in the order of `preaction` options given.

In the following example, we use one `preaction` to add a shadow and another to provide a shading, while the main action is to use a pattern.



```
\usetikzlibrary {patterns}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw [pattern=fivepointed stars]
    [preaction={fill=black,opacity=.5,
                transform canvas={xshift=1mm,yshift=-1mm}}]
    [preaction={top color=blue,bottom color=white}]
               (0,0) rectangle (1,2)
               (1,2) circle [radius=5mm];
\end{tikzpicture}
```

A complicated application is shown in the following example, where the path is used several times with different fadings and shadings to create a special visual effect:




```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  [
    % Define an interesting style
    button/.style={
      % First preaction: Fuzzy shadow
      preaction={fill=black,path fading=circle with fuzzy edge 20 percent,
                 opacity=.5,transform canvas={xshift=1mm,yshift=-1mm}},
      % Second preaction: Background pattern
      preaction={pattern=#1,
                 path fading=circle with fuzzy edge 15 percent},
      % Third preaction: Make background shiny
      preaction={top color=white,
                 bottom color=black!50,
                 shading angle=45,
                 path fading=circle with fuzzy edge 15 percent,
                 opacity=0.2},
      % Fourth preaction: Make edge especially shiny
      preaction={path fading=fuzzy ring 15 percent,
                 top color=black!5,
                 bottom color=black!80,
                 shading angle=45},
      inner sep=2ex
    },
    button/.default=horizontal lines light blue,
    circle
  ]

  \draw [help lines] (0,0) grid (4,3);

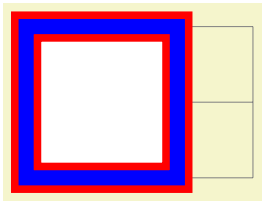
  \node [button] at (2.2,1) {\Huge Big};
  \node [button=crosshatch dots light steel blue,
         text=white] at (1,1.5) {Small};
\end{tikzpicture}
```

/tikz/postaction=⟨*options*⟩ (no default)

The postactions work in the same way as the preactions, only they are applied *after* the main action has been taken. Like preactions, multiple `postaction` options may be given to a `\path` command, in which case the path is reused several times, each time with a different set of options in force.

If both pre- and postactions are specified, then the preactions are taken first, then the main action, and then the post actions.

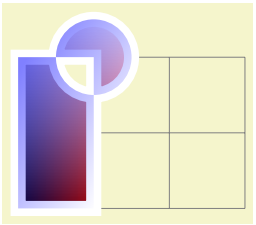
In the first example, we use a postaction to draw the path, after it has already been drawn:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \draw
    [postaction={draw,line width=2mm,blue}]
    [line width=4mm,red,fill=white] (0,0) rectangle (2,2);
\end{tikzpicture}
```

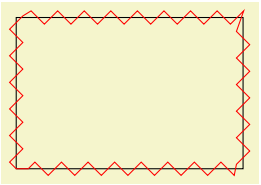
In another example, we use a postaction to "colorize" a path:



```
\usetikzlibrary {fadings}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw
    [postaction={path fading=south,fill=white}]
    [postaction={path fading=south,fading angle=45,fill=blue,opacity=.5}]
    [left color=black,right color=red,draw=white,line width=2mm]
                (0,0) rectangle (1,2)
                (1,2) circle [radius=5mm];
\end{tikzpicture}
```

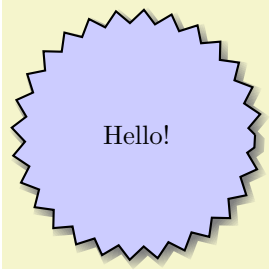
15.11 Decorating and Morphing a Path

Before a path is used, it is possible to first "decorate" and/or "morph" it. Morphing means that the path is replaced by another path that is slightly varied. Such morphings are a special case of the more general "decorations" described in detail in Section 24. For instance, in the following example the path is drawn twice: Once normally and then in a morphed (=decorated) manner.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw (0,0) rectangle (3,2);
  \draw [red, decorate, decoration=zigzag]
        (0,0) rectangle (3,2);
\end{tikzpicture}
```

Naturally, we could have combined this into a single command using pre- or postaction. It is also possible to deform shapes:

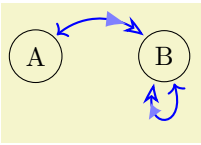



```
\usetikzlibrary {decorations.pathmorphing,shadows}
\begin{tikzpicture}
  \node [circular drop shadow={shadow scale=1.05},minimum size=3.13cm,
         decorate, decoration=zigzag,
         fill=blue!20,draw,thick,circle] {Hello!};
\end{tikzpicture}
```

16 Arrows

16.1 Overview

Ti*k*Z allows you to add (multiple) arrow tips to the end of lines as in ↠ or in ⟶. It is possible to change which arrow tips are used "on-the-fly", you can have several arrow tips in a row, and you can change the appearance of each of them individually using a special syntax. The following example is a perhaps slightly "excessive" demonstration of what you can do (you need to load the `arrows.meta` library for it to work):




```
\usetikzlibrary {arrows.meta,bending,positioning}
\tikz {
  \node [circle,draw] (A)              {A};
  \node [circle,draw] (B) [right=of A] {B};

  \draw [draw = blue, thick,
         arrows={
           Computer Modern Rightarrow [sep]
         - Latex[blue!50,length=8pt,bend,line width=0pt]
           Stealth[length=8pt,open,bend,sep]}]
    (A) edge [bend left=45] (B)
    (B) edge [in=-110, out=-70,looseness=8] (B);
}
```

There are a number of predefined generic arrow tip kinds whose appearance you can modify in many ways using various options. It is also possible to define completely new arrow tip kinds, see Section 105, but doing this is somewhat harder than configuring an existing kind (it is like the difference between using a font at different sizes or faces like italics, compared to designing a new font yourself).

In the present section, we go over the various ways in which you can configure which particular arrow tips are *used*. The glorious details of how new arrow tips can be defined are explained in Section 105.

At the end of the present section, Section 16.5, you will find a description of the different predefined arrow tips from the `arrows.meta` library.

Remark: Almost all of the features described in the following were introduced in version 3.0 of Ti*k*Z. For compatibility reasons, the old arrow tips are still available. To differentiate between the old and new arrow tips, the following rule is used: The new, more powerful arrow tips start with an uppercase letter as in `Latex`, compared to the old arrow tip `latex`.

Remark: The libraries `arrows` and `arrows.spaced` are deprecated. Use `arrows.meta` instead/additionally, which allows you to do all that the old libraries offered, plus much more. However, the old libraries still work and you can even mix old and new arrow tips (only, the old arrow tips cannot be configured in the ways described in the rest of this section; saying `scale=2` for a `latex` arrow has no effect for instance, while for `Latex` arrows it doubles their size as one would expect.)

16.2 Where and When Arrow Tips Are Placed

In order to add arrow tips to the lines you draw, the following conditions must be met:

1. You have specified that arrow tips should be added to lines, using the `arrows` key or its short form.
2. You set the `tips` key to some value that causes tips to be drawn (to be explained later).
3. You do not use the `clip` key (directly or indirectly) with the current path.
4. The path actually has two "end points" (it is not "closed").

Let us start with an introduction to the basics of the `arrows` key:

`/tikz/arrows=`⟨*start arrow specification*⟩`-`⟨*end arrow specification*⟩ (no default)

This option sets the arrow tip(s) to be used at the start and end of lines. An empty value as in `->` for the start indicates that no arrow tip should be drawn at the start.

Note: Since the arrow option is so often used, you can leave out the text `arrows=`. What happens is that every (otherwise unknown) option that contains a `-` is interpreted as an arrow specification.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
  \draw[->]          (0,0)   -- (1,0);
  \draw[>-Stealth] (0,0.3) -- (1,0.3);
\end{tikzpicture}
```

In the above example, the first start specification is empty and the second is `>`. The end specifications are `>` for the first line and `Stealth` for the second line. Note that it makes a difference whether `>` is used in a start specification or in an end specification: In an end specification it creates, as one would expect, a pointed tip at the end of the line. In the start specification, however, it creates a "reversed" version if this arrow – which happens to be what one would expect here.

The above specifications are very simple and only select a single arrow tip without any special configuration options, resulting in the "natural" versions of these arrow tips. It is also possible to "configure" arrow tips in many different ways, as explained in detail in Section 16.3 below by adding options in square brackets following the arrow tip kind:


```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
  \draw[-{Stealth[red]}] (0,0)   -- (1,0);
\end{tikzpicture}
```

Note that in the example I had to surround the end specification by braces. This is necessary so that Ti*k*Z does not mistake the closing square bracket of the `Stealth` arrow tip's options for the end of the options of the `\draw` command. In general, you often need to add braces when specifying arrow tips except for simple case like `->` or `<<->`, which are pretty frequent, though. When in doubt, say `arrows={`⟨*start spec*⟩`-`⟨*end spec*⟩`}`, that will always work.

It is also possible to specify multiple (different) arrow tips in a row inside a specification, see Section 16.4 below for details.

As was pointed out earlier, to add arrow tips to a path, the path must have "end points" and not be "closed" – otherwise adding arrow tips makes little sense, after all. However, a path can actually consist of several subpath, which may be open or not and may even consist of only a single point (a single move-to). In this case, it is not immediately obvious, where arrow heads should be placed. The actual rules that Ti*k*Z uses are governed by the setting of the key `tips`:

`/pgf/tips=`⟨*value*⟩ (default `true`, initially `on draw`)

alias `/tikz/tips`

This key governs in what situations arrow tips are added to a path. The following ⟨*values*⟩ are permissible:

- `true` (the value used when no ⟨*value*⟩ is specified)
- `proper`
- `on draw` (the initial value, if the key has not yet been used at all)
- `on proper draw`
- `never` or `false` (same effect)

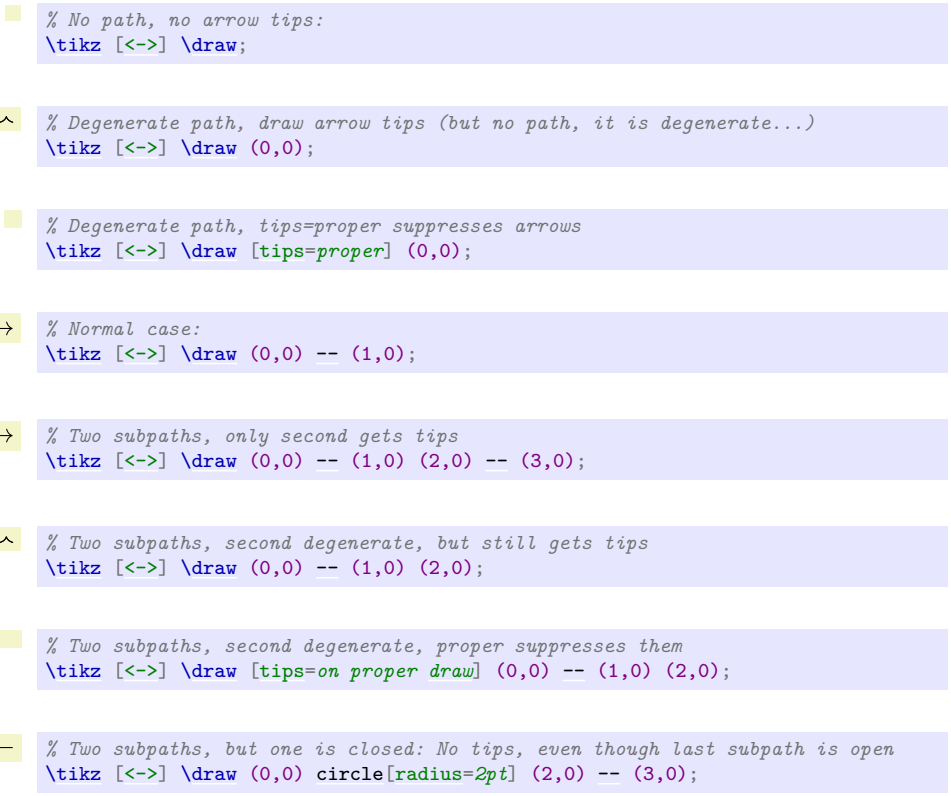
Firstly, there are a whole bunch of situations where the setting of these (or other) options causes no arrow tips to be shown:

- If no arrow tips have been specified (for instance, by having said `arrows=-`), no arrow tips are drawn.
- If the `clip` option is set, no arrow tips are drawn.
- If `tips` has been set to `never` or `false`, no arrow tips are drawn.
- If `tips` has been set to `on draw` or `on proper draw`, but the `draw` option is not set, no arrow tips are drawn.
- If the path is empty (as in `\path ;`), no arrow tips are drawn.
- If at least one of the subpaths of a path is closed (`cycle` is used somewhere or something like `circle` or `rectangle`), arrow tips are never drawn anywhere – even if there are open subpaths.

Now, if we pass all of the above tests, we must have a closer look at the path. All its subpaths must now be open and there must be at least one subpath. We consider the last one. Arrow tips will only be added to this last subpath.

1. If this last subpath not degenerate (all coordinates on the subpath are the same as in a single "move-to" `\path (0,0);` or in a "move-to" followed by a "line-to" to the same position as in `\path (1,2) -- (1,2)`), arrow tips are added to this last subpath now.

2. If the last subpath is degenerate, we add arrow tips pointing upward at the single coordinate mentioned in the path, but only for `tips` begin set to `true` or to `on draw` – and not for `proper` nor for `on proper draw`. In other words, "proper" suppresses arrow tips on degenerate paths.



```
% No path, no arrow tips:
\tikz [<->] \draw;

% Degenerate path, draw arrow tips (but no path, it is degenerate...)
\tikz [<->] \draw (0,0);

% Degenerate path, tips=proper suppresses arrows
\tikz [<->] \draw [tips=proper] (0,0);

% Normal case:
\tikz [<->] \draw (0,0) -- (1,0);

% Two subpaths, only second gets tips
\tikz [<->] \draw (0,0) -- (1,0) (2,0) -- (3,0);

% Two subpaths, second degenerate, but still gets tips
\tikz [<->] \draw (0,0) -- (1,0) (2,0);

% Two subpaths, second degenerate, proper suppresses them
\tikz [<->] \draw [tips=on proper draw] (0,0) -- (1,0) (2,0);

% Two subpaths, but one is closed: No tips, even though last subpath is open
\tikz [<->] \draw (0,0) circle[radius=2pt] (2,0) -- (3,0);
```

One common pitfall when arrow tips are added to a path should be addressed right here at the beginning: When Ti*k*Z positions an arrow tip at the start, for all its computations it only takes into account the first segment of the subpath to which the arrow tip is added. This "first segment" is the first line-to or curve-to operation (or arc or parabola or a similar operation) of the path; but note that decorations like `snake` will add many small line segments to paths. The important point is that if this first segment is very small, namely smaller that the arrow tip itself, strange things may result. As will be explained in Section 16.3.8, Ti*k*Z will modify the path by shortening the first segment and shortening a segment below its length may result in strange effects. Similarly, for tips at the end of a subpath, only the last segment is considered.

The bottom line is that wherever an arrow tip is added to a path, the line segment where it is added should be "long enough".

16.3 Arrow Keys: Configuring the Appearance of a Single Arrow Tip

For standard arrow tip kinds, like `Stealth` or `Latex` or `Bar`, you can easily change their size, aspect ratio, color, and other parameters. This is similar to selecting a font face from a font family: *"This text"* is not just typeset in the font "Computer Modern", but rather in "Computer Modern, italic face, 11pt size, medium weight, black color, no underline, …" Similarly, an arrow tip is not just a "Stealth" arrow tip, but rather a "Stealth arrow tip at its natural size, flexing, but not bending along the path, miter line caps, draw and fill colors identical to the path draw color, …"

Just as most programs make it easy to "configure" which font should be used at a certain point in a text, Ti*k*Z tries to make it easy to specify which configuration of an arrow tip should be used. You use *arrow keys*, where a certain parameter like the `length` of an arrow is set to a given value using the standard key–value syntax. You can provide several arrow keys following an arrow tip kind in an arrow tip specification as in `Stealth[length=4pt,width=2pt]`.

While selecting a font may be easy, *designing* a new font is a highly creative and difficult process and more often than not, not all faces of a font are available on any given system. The difficulties involved in designing a new arrow tip are somewhat similar to designing a new letter for a font and, thus, it may also happen that not all configuration options are actually implemented for a given arrow tip. Naturally, for the

standard arrow tips, all configuration options are available – but for special-purpose arrow tips it may well happen that an arrow tip kind simply "ignores" some of the configurations given by you.

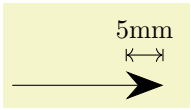
Some of the keys explained in the following are defined in the library `arrows.meta`, others are always available. This has to do with the question of whether the arrow key needs to be supported directly in the PGF core or not. In general, the following explanations assume that `arrows.meta` has been loaded.

16.3.1 Size

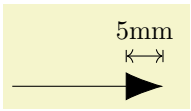
The most important configuration parameter of an arrow tip is undoubtedly its size. The following two keys are the main keys that are important in this context:

`/pgf/arrow keys/length`=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

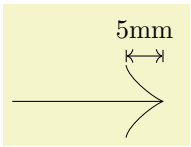
This parameter is usually the most important parameter that governs the size of an arrow tip: The ⟨*dimension*⟩ that you provide dictates the distance from the "very tip" of the arrow to its "back end" along the line:




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Stealth[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
}
```




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Latex[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.4) -- node[above=1mm] {5mm} (2,.4);
}
```

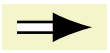



```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [-{Classical TikZ Rightarrow[length=5mm]}] (0,0) -- (2,0);
  \draw [|<->|] (1.5,.6) -- node[above=1mm] {5mm} (2,.6);
}
```

The Line Width Factors. Following the ⟨*dimension*⟩, you may put a space followed by a ⟨*line width factor*⟩, which must be a plain number (no `pt` or `cm` following). When you provide such a number, the size of the arrow tip is not just ⟨*dimension*⟩, but rather ⟨*dimension*⟩ + ⟨*line width factor*⟩ · w where w is the width of the to-be-drawn path. This makes it easy to vary the size of an arrow tip in accordance with the line width – usually a very good idea since thicker lines will need thicker arrow tips.

As an example, when you write `length=0pt 5`, the length of the arrow will be exactly five times the current line width. As another example, the default length of a `Latex` arrow is `length=3pt 4.5 0.8`. Let us ignore the 0.8 for a moment; the `3pt 4.5` then means that for the standard line width of `0.4pt`, the length of a `Latex` arrow will be exactly 4.8pt (3pt plus 4.5 times `0.4pt`).

Following the line width factor, you can additionally provide an ⟨*outer factor*⟩, again preceded by a space (the `0.8` in the above example). This factor is taken into consideration only when the `double` option is used, that is, when a so-called "inner line width". For a double line, we can identify three different "line widths", namely the inner line width w_i, the line width w_o of the two outer lines, and the "total line width" $w_t = w_i + 2w_o$. In the below examples, we have $w_i = 3\text{pt}$, $w_o = 1\text{pt}$, and $w_t = 5\text{pt}$. It is not immediately clear which of these line widths should be considered as w in the above formula ⟨*dimension*⟩ + ⟨*line width factor*⟩ · w for the computation of the length. One can argue both for w_t and also for w_o. Because of this, you use the ⟨*outer factor*⟩ to decide on one of them or even mix them: TikZ sets $w = \langle\textit{outer factor}\rangle w_o + (1 - \langle\textit{outer factor}\rangle)w_t$. Thus, when the outer factor is 0, as in the first of the following examples and as is the default when it is not specified, the computed w will be the total line width $w_t = 5\text{pt}$. Since $w = 5\text{pt}$, we get a total length of $15pt$ in the first example (because of the factor `3`). In contrast, in the last example, the outer factor is 1 and, thus, $w = w_o = 1\text{pt}$ and the resulting length is 3pt. Finally, for the middle case, the "middle" between 5pt and 1pt is 3pt, so the length is 9pt.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 0]}] (0,0) -- (1,0);
```



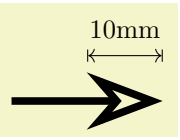
```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 .5]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1pt, double distance=3pt,
             arrows = {-Latex[length=0pt 3 1]} ] (0,0) -- (1,0);
```

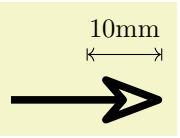
The Exact Length. For an arrow tip kind that is just an outline that is filled with a color, the specified length should *exactly* equal the distance from the tip to the back end. However, when the arrow tip is drawn by stroking a line, it is no longer obvious whether the `length` should refer to the extend of the stroked lines' path or of the resulting pixels (which will be wider because of the thickness of the stroking pen). The rules are as follows:

1. If the arrow tip consists of a closed path (like `Stealth` or `Latex`), imagine the arrow tip drawn from left to right using a miter line cap. Then the `length` should be the horizontal distance from the first drawn "pixel" to the last drawn "pixel". Thus, the thickness of the stroked line and also the miter ends should be taken into account:




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open]}]
          (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
}
```

2. If, in the above case, the arrow is drawn using a round line join (see Section 16.3.7 for details on how to select this), the size of the arrow should still be the same as in the first case (that is, as if a miter join were used). This creates some "visual consistency" if the two modes are mixed or if you later want to change the mode.




```
\usetikzlibrary {arrows.meta}
\tikz{
  \draw [line width=1mm, -{Stealth[length=10mm, open, round]}]
          (0,0) -- (2,0);
  \draw [|<->|] (2,.6) -- node[above=1mm] {10mm} ++(-10mm,0);
}
```

As the above example shows, however, a rounded arrow will still exactly "tip" the point where the line should end (the point `(2,0)` in the above case). It is only the scaling of the arrow that is not affected.

`/pgf/arrow keys/width`=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

This key works like the `length` key, only it specifies the "width" of the arrow tip; so if width and length are identical, the arrow will just touch the borders of a square. (An exception to this rule are "halved" arrow tips, see Section 16.3.5.) The meaning of the two optional factor numbers following the ⟨*dimension*⟩ is the same as for the `length` key.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width=10pt, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width=0pt 10, length=10pt]}] (0,0) -- (1,0);
```

`/pgf/arrow keys/width'`=⟨*dimension*⟩ ⟨*length factor*⟩ ⟨*line width factor*⟩ (no default)

The key (note the prime) has a similar effect as the `width` key. The difference is that the second, still optional parameter ⟨*length factor*⟩ specifies the width of the key not as a multiple of the line width, but as a multiple of the arrow length.

The idea is that if you write, say, `width'=0pt 0.5`, the width of the arrow will be half its length. Indeed, for standard arrow tips like `Stealth` the default width is specified in this way so that if you change the length of an arrow tip, you also change the width in such a way that the aspect ratio of the arrow tip is kept. The other way round, if you modify the factor in `width'` without changing the length, you change the aspect ratio of the arrow tip.

Note that later changes of the length are taken into account for the computation. For instance, if you write

```
length = 10pt, width'=5pt 2, length=7pt
```

the resulting width will be $19\text{pt} = 5\text{pt} + 2 \cdot 7\text{pt}$.


```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width'=0pt .5, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[width'=0pt .5, length=15pt]}] (0,0) -- (1,0);
```

The third, also optional, parameter allows you to add a multiple of the line width to the value computed in terms of the length.

`/pgf/arrow keys/inset`=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

The key is relevant only for some arrow tips such as the `Stealth` arrow tip. It specifies a distance by which something inside the arrow tip is set inwards; for the `Stealth` arrow tip it is the distance by which the back angle is moved inwards.

The computation of the distance works in the same way as for `length` and `width`: To the ⟨*dimension*⟩ we add ⟨*line width factor*⟩ times that line width, where the line width is computed based on the ⟨*outer factor*⟩ as described for the `length` key.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[length=10pt, inset=5pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[length=10pt, inset=2pt]}] (0,0) -- (1,0);
```

For most arrows for which there is no "natural inset" like, say, `Latex`, this key has no effect.

`/pgf/arrow keys/inset'`=⟨*dimension*⟩ ⟨*length factor*⟩ ⟨*line width factor*⟩ (no default)

This key works like `inset`, only like `width'` the second parameter is a factor of the arrow length rather than of the line width. For instance, the `Stealth` arrow sets `inset'` to `0pt 0.325` to ensure that the inset is always at 13/40th of the arrow length if nothing else is specified.

`/pgf/arrow keys/angle`=⟨*angle*⟩:⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

This key sets the `length` and the `width` of an arrow tip at the same time. The length will be the cosine of ⟨*angle*⟩, while the width will be twice the sine of half the ⟨*angle*⟩ (this slightly awkward rule ensures that a `Stealth` arrow will have an opening angle of ⟨*angle*⟩ at its tip if this option is used). As for the `length` key, if the optional factors are given, they add a certain multiple of the line width to the ⟨*dimension*⟩ before the sine and cosines are computed.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, angle=90:10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, angle=30:10pt]}] (0,0) -- (1,0);
```

`/pgf/arrow keys/angle'`=⟨*angle*⟩ (no default)

Sets the width of the arrow to twice the tangent of ⟨*angle*⟩/2 times the arrow length. This results in an arrow tip with an opening angle of ⟨*angle*⟩ at its tip and with the specified `length` unchanged.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=90]}]
            (0,0) -- (1,0);
```



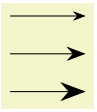
```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Stealth[inset=0pt, length=10pt, angle'=30]}]
            (0,0) -- (1,0);
```

16.3.2 Scaling

In the previous section we saw that there are many options for getting "fine control" overt the length and width of arrow tips. However, in some cases, you do not really care whether the arrow tip is 4pt long or 4.2pt long, you "just want it to be a little bit larger than usual". In such cases, the following keys are useful:

`/pgf/arrow keys/scale`=⟨*factor*⟩ (no default, initially `1`)

After all the other options listed in the previous (and also the following sections) have been processed, Ti*k*Z applies a *scaling* to the computed length, inset, and width of the arrow tip (and, possibly, to other size parameters defined by special-purpose arrow tip kinds). Everything is simply scaled by the given ⟨*factor*⟩.



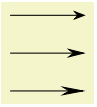
```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {-Stealth[]}]           (0,1)   -- (1,1);
  \draw [arrows = {-Stealth[scale=1.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {-Stealth[scale=2]}]    (0,0)   -- (1,0);
}
```

Note that scaling has *no* effect on the line width (as usual) and also not on the arrow padding (the `sep`).

You can get even more fine-grained control over scaling using the following keys (the `scale` key is just a shorthand for setting both of the following keys simultaneously):

`/pgf/arrow keys/scale length`=⟨*factor*⟩ (no default, initially `1`)

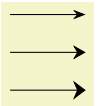
This factor works like `scale`, only it is applied only to dimensions "along the axis of the arrow", that is, to the length and to the inset, but not to the width.



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {-Stealth[]}]                  (0,1)   -- (1,1);
  \draw [arrows = {-Stealth[scale length=1.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {-Stealth[scale length=2]}]    (0,0)   -- (1,0);
}
```

`/pgf/arrow keys/scale width`=⟨*factor*⟩ (no default, initially `1`)

Like `scale length`, but for dimensions related to the width.



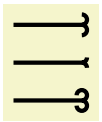
```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {-Stealth[]}]                  (0,1)   -- (1,1);
  \draw [arrows = {-Stealth[scale width=1.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {-Stealth[scale width=2]}]    (0,0)   -- (1,0);
}
```

16.3.3 Arc Angles

A few arrow tips consist mainly of arcs, whose length can be specified. For these arrow tips, you use the following key:

`/pgf/arrow keys/arc`=⟨*degrees*⟩ (no default, initially `180`)

Sets the angle of arcs in arrows to ⟨*degrees*⟩. Note that this key is quite different from the `angle` key, which is "just a fancy way of setting the length and width". In contrast, the `arc` key is used to set the degrees of arcs that are part of an arrow tip:



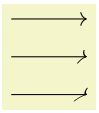
```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] {
  \draw [arrows = {-Hooks[]}]          (0,1)   -- (1,1);
  \draw [arrows = {-Hooks[arc=90]}]    (0,0.5) -- (1,0.5);
  \draw [arrows = {-Hooks[arc=270]}]   (0,0)   -- (1,0);
}
```

16.3.4 Slanting

You can "slant" arrow tips using the following key:

`/pgf/arrow keys/slant`=⟨*factor*⟩ (no default, initially 0)

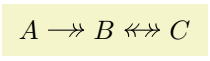
Slanting is used to create an "italics" effect for arrow tips: All arrow tips get "slanted" a little bit relative to the axis of the arrow:



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [arrows = {->[]}]          (0,1)   -- (1,1);
  \draw [arrows = {->[slant=.5]}] (0,0.5) -- (1,0.5);
  \draw [arrows = {->[slant=1]}]   (0,0)   -- (1,0);
}
```

There is one thing to note about slanting: Slanting is done using a so-called "canvas transformation" and has no effect on positioning of the arrow tip. In particular, if an arrow tip gets slanted so strongly that it starts to protrude over the arrow tip end, this does not change the positioning of the arrow tip.

Here is another example where slanting is used to match italic text:



```
\usetikzlibrary {arrows.meta,graphs}
\tikz [>={[slant=.3] To[] To[]}]
  \graph [math nodes] { A -> B <-> C };
```

16.3.5 Reversing, Halving, Swapping

`/pgf/arrow keys/reversed` (no value)

Adding this key to an arrow tip will "reverse its direction" so that is points in the opposite direction (but is still at that end of the line where the non-reversed arrow tip would have been drawn; so only the tip is reversed). For most arrow tips, this just results in an internal flip of a coordinate system, but some arrow tips actually use a slightly different version of the tip for reversed arrow tips (namely when the joining of the tip with the line would look strange). All of this happens automatically, so you do not need to worry about this.

If you apply this key twice, the effect cancels, which is useful for the definition of shorthands (which will be discussed later).



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[reversed]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[reversed, reversed]}] (0,0) -- (1,0);
```

`/pgf/arrow keys/harpoon` (no value)

The key requests that only the "left half" of the arrow tip should drawn:


```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {->[harpoon]}] (0,0) -- (1,0);
```

Unlike the `reversed` key, which all arrows tip kinds support at least in a basic way, designers of arrow tips really need to take this key into account in their arrow tip code and often a lot of special attention needs to do be paid to this key in the implementation. For this reason, only some arrow tips will support it.

/pgf/arrow keys/swap (no value)

This key flips that arrow tip along the axis of the line. It makes sense only for asymmetric arrow tips like the harpoons created using the `harpoon` option.


```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[harpoon,swap]}] (0,0) -- (1,0);
```

Swapping is always possible, no special code is needed on behalf of an arrow tip implementer.

/pgf/arrow keys/left (no value)

A shorthand for `harpoon`.

/pgf/arrow keys/right (no value)

A shorthand for `harpoon, swap`.


```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[left]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [arrows = {-Stealth[right]}] (0,0) -- (1,0);
```

16.3.6 Coloring

Arrow tips are drawn using the same basic mechanisms as normal paths, so arrow tips can be stroked (drawn) and/or filled. However, we usually want the color of arrow tips to be identical to the color used to draw the path, even if a different color is used for filling the path. On the other hand, we may also sometimes wish to use a special color for the arrow tips that is different from both the line and fill colors of the main path.

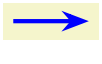
The following options allow you to configure how arrow tips are colored:

/pgf/arrow keys/color=⟨*color or empty*⟩ (no default, initially empty)

Normally, an arrow tip gets the same color as the path to which it is attached. More precisely, it will get the current "draw color", also known as "stroke color", which you can set using `draw=`⟨*some color*⟩. By adding the option `color=` to an arrow tip (note that an "empty" color is specified in this way), you ask that the arrow tip gets this default draw color of the path. Since this is the default behavior, you usually do not need to specify anything:

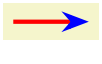


```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth}] (0,0) -- (1,0);
```

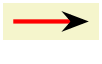


```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [blue, arrows = {-Stealth}] (0,0) -- (1,0);
```

Now, when you provide a ⟨*color*⟩ with this option, you request that the arrow tip should get this color *instead* of the color of the main path:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[color=blue]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[color=black]}] (0,0) -- (1,0);
```

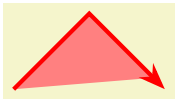
Similar to the `color` option used in normal Ti*k*Z options, you may omit the `color=` part of the option. Whenever an ⟨*arrow key*⟩ is encountered that Ti*k*Z does not recognize, it will test whether the key is the name of a color and, if so, execute `color=`⟨*arrow key*⟩. So, the first of the above examples can be rewritten as follows:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [red, arrows = {-Stealth[blue]}] (0,0) -- (1,0);
```

The ⟨*color*⟩ will apply both to any drawing and filling operations used to construct the path. For instance, even though the `Stealth` arrow tips looks like a filled quadrilateral, it is actually constructed by drawing a quadrilateral and then filling it in the same color as the drawing (see the `fill` option below to see the difference).

When `color` is set to an empty text, the drawing color is always used to fill the arrow tips, even if a different color is specified for filling the path:



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [draw=red, fill=red!50, arrows = {-Stealth[length=10pt]}]
                          (0,0) -- (1,1) -- (2,0);
```

As you can see in the above example, the filled area is not quite what you might have expected. The reason is that the path was actually internally shortened a bit so that the end of the "fat line" as inside the arrow tip and we get a "clear" arrow tip.

In general, it is a good idea not to add arrow tips to paths that are filled.

`/pgf/arrow keys/fill=`⟨*color*⟩ (default is scope's color setting)

Use this key to explicitly set the color used for filling the arrow tips. This color can be different from the color used to draw (stroke) the arrow tip:



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
  \draw [thick, red, arrows = {-Stealth[fill=white,length=15pt]}] (0,0) -- (1,0);
}
```

You can also specify the special "color" `none`. In this case, the arrow tip is not filled at all (not even with white):



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
  \draw [thick, red, arrows = {-Stealth[fill=none,length=15pt]}] (0,0) -- (1,0);
}
```

If the argument is left empty or omitted altogether, no explicit fill color will be set, meaning that the fill color will be determined by whatever was used last as a fill color in the current scope.

Note that such "open" arrow tips are a bit difficult to draw in some case: The problem is that the line must be shortened by just the right amount so that it ends exactly on the back end of the arrow tip. In some cases, especially when double lines are used, this will not be possible.

`/pgf/arrow keys/open` (no value)

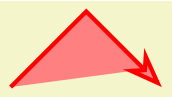
A shorthand for `fill=none`.

When you use both the `color` and `fill` options, the `color` option must come first since it will reset the filling to the color specified for drawing.



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [help lines] (0,-.5) grid [step=1mm] (1,.5);
  \draw [thick, red, arrows = {-Stealth[color=blue, fill=white, length=15pt]}]
        (0,0) -- (1,0);
}
```

Note that by setting `fill` to the special color `pgffillcolor`, you can cause the arrow tips to be filled using the color used to fill the main path. (This special color is always available and always set to the current filling color of the graphic state.):



```
\usetikzlibrary {arrows.meta}
\tikz [ultra thick] \draw [draw=red, fill=red!50,
                           arrows = {-Stealth[length=15pt, fill=pgffillcolor]}]
                          (0,0) -- (1,1) -- (2,0);
```

16.3.7 Line Styling

Arrow tips are created by drawing and possibly filling a path that makes up the arrow tip. When TikZ draws a path, there are different ways in which such a path can be drawn (such as dashing). Three particularly important parameters are the line join, the line cap, see Section 15.3.1 for an introduction, and the line width (thickness).

TikZ resets the line cap and line join each time it draws an arrow tip since you usually do not want their settings to "spill over" to the way the arrow tips are drawn. You can, however, change there values explicitly for an arrow tip:

`/pgf/arrow keys/line cap=`⟨*round* or *butt*⟩ (no default)

Sets the line cap of all lines that are drawn in the arrow to a round cap or a butt cap. (Unlike for normal lines, the `rect` cap is not allowed.) Naturally, this key has no effect for arrows whose paths are closed.

Each arrow tip has a default value for the line cap, which can be overruled using this option.

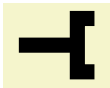
Changing the cap should have no effect on the size of the arrow. However, it will have an effect on where the exact "tip" of the arrow is since this will always be exactly at the end of the arrow:



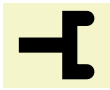
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line cap=butt]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line cap=round]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=butt]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line cap=round]}]
        (0,0) -- (1,0);
```

`/pgf/arrow keys/line join=`⟨*round* or *miter*⟩ (no default)

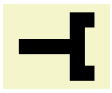
Sets the line join to round or miter (`bevel` is not allowed). This time, the key only has an effect on paths that have "corners" in them. The same rules as for `line cap` apply: the size is not affects, but the tip end is:



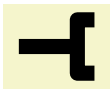
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=miter]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[line join=round]}]
        (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line join=miter]}]
        (0,0) -- (1,0);
```

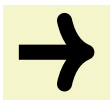


```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,line join=round]}]
        (0,0) -- (1,0);
```

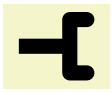
The following keys set both of the above:

/pgf/arrow keys/round (no value)

A shorthand for `line cap=round, line join=round`, resulting in "rounded" arrow heads.



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[round]}] (0,0) -- (1,0);
```



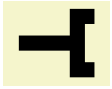
```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,round]}] (0,0) -- (1,0);
```

/pgf/arrow keys/sharp (no value)

A shorthand for `line cap=butt, line join=miter`, resulting in "sharp" or "pointed" arrow heads.



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Computer Modern Rightarrow[sharp]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [line width=2mm]
  \draw [arrows = {-Bracket[reversed,sharp]}] (0,0) -- (1,0);
```

You can also set the width of lines used inside arrow tips:

/pgf/arrow keys/line width=⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (no default)

This key sets the line width inside an arrow tip for drawing (out)lines of the arrow tip. When you set this width to `0pt`, which makes sense only for closed tips, the arrow tip is only filled. This can result in better rendering of some small arrow tips and in case of bend arrow tips (because the line joins will also be bend and not "mitered".)

The meaning of the factors is as usual the same as for `length` or `width`.



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[line width=0.1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [arrows = {-Latex[line width=1pt, fill=white, length=10pt]}] (0,0) -- (1,0);
```

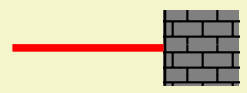
/pgf/arrow keys/line width'=⟨*dimension*⟩ ⟨*length factor*⟩ (no default)

Works like `line width` only the factor is with respect to the `length`.

16.3.8 Bending and Flexing

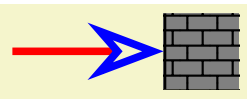
Up to now, we have only added arrow tip to the end of straight lines, which is in some sense "easy". Things get far more difficult, if the line to which we wish to end an arrow tip is curved. In the following, we have a look at the different actions that can be taken and how they can be configured.

To get a feeling for the difficulties involved, consider the following situation: We have a "gray wall" at the x-coordinate of and a red line that ends in its middle.



```
\usetikzlibrary {patterns}
\def\wall{ \fill     [fill=black!50]  (1,-.5) rectangle (2,.5);
           \pattern  [pattern=bricks] (1,-.5) rectangle (2,.5);
           \draw     [line width=1pt]  (1cm+.5pt,-.5) -- ++(0,1); }
\begin{tikzpicture}
  \wall
  % The "line"
  \draw [red,line width=1mm] (-1,0) -- (1,0);
\end{tikzpicture}
```

Now we wish to add a blue open arrow tip the red line like, say, `Stealth[length=1cm,open,blue]`:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
  \wall
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue]}]
        (-1,0) -- (1,0);
\end{tikzpicture}
```

There are several noteworthy things about the blue arrow tip:

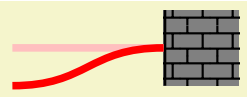
1. Notice that the red line no longer goes all the way to the wall. Indeed, the red line ends more or less exactly where it meets the blue line, leaving the arrow tip empty. Now, recall that the red line was supposed to be the path `(-2,0)--(1,0)`; however, this path has obviously become much shorter (by 6.25mm to be precise). This effect is called *path shortening* in Ti*k*Z.

2. The very tip of the arrow just "touches" the wall, even we zoom out a lot. This point, where the original path ended and where the arrow tip should now lie, is called the *tip end* in Ti*k*Z.

3. Finally, the point where the red line touches the blue line is the point where the original path "visually ends". Notice that this is not the same as the point that lies at a distance of the arrow's `length` from the wall – rather it lies at a distance of `length` minus the `inset`. Let us call this point the *visual end* of the arrow.

As pointed out earlier, for straight lines, shortening the path and rotating and shifting the arrow tip so that it ends precisely at the tip end and the visual end lies on a line from the tip end to the start of the line is relatively easy.

For curved lines, things are much more difficult and Ti*k*Z copes with the difficulties in different ways, depending on which options you add to arrows. Here is now a curved red line to which we wish to add our arrow tip (the original straight red line is shown in light red):



```
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm] (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

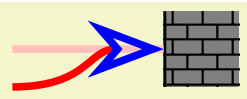
The first way of dealing with curved lines is dubbed the "quick and dirty" way (although the option for selecting this option is politely just called "`quick`" …):

`/pgf/arrow keys/quick` (no value)

Recall that curves in Ti*k*Z are actually Bézier curves, which means that they start and end at certain points and we specify two vectors, one for the start and one for the end, that provide tangents to the curve at these points. In particular, for the end of the curve, there is a point called the *second support point* of the curve such that a tangent to the curve at the end goes through this point. In our above example, the second support point is at the middle of the light red line and, indeed, a tangent to the red line at the point touching the wall is perfectly horizontal.

In order to add our arrow tip to the curved path, our first objective is to "shorten" the path by 6.25mm. Unfortunately, this is now much more difficult than for a straight path. When the `quick` option is added

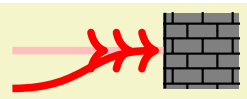
to an arrow tip (it is also the default if no special libraries are loaded), we cheat somewhat: Instead of really moving along 6.25mm along the path, we simply shift the end of the curve by 6.25mm *along the tangent* (which is easy to compute). We also have to shift the second support point by the same amount to ensure that the line still has the same tangent at the end. This will result in the following:



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,quick]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

They main problem with the above picture is that the red line is no longer equal to the original red line (notice much sharper curvature near its end). In our example this is not such a bad thing, but it certainly "not a nice thing" that adding arrow tips to a curve changes the overall shape of the curves. This is especially bothersome if there are several similar curves that have different arrow heads. In this case, the similar curves now suddenly look different.

Another big problem with the above approach is that it works only well if there is only a single arrow tip. When there are multiple ones, simply shifting them along the tangent as the `quick` option does produces less-than-satisfactory results:



```
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{[quick,sep]>>>}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Note that the third arrow tip does not really lie on the curve any more.

Because of the shortcomings of the `quick` key, more powerful mechanisms for shortening lines and rotating arrows tips have been implemented. To use them, you need to load the following library:

TikZ Library bending

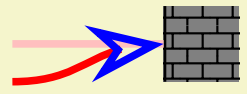
```
\usetikzlibrary{bending} % LaTeX and plain TeX
\usetikzlibrary[bending] % ConTeXt
```

Load this library to use the `flex`, `flex'`, or `bending` arrow keys. When this library is loaded, `flex` becomes the default mode that is used with all paths, unless `quick` is explicitly selected for the arrow tip.

`/pgf/arrow keys/flex=`⟨*factor*⟩ (default 1)

When the `bending` library is loaded, this key is applied to all arrow tips by default. It has the following effect:

1. Instead of simply shifting the visual end of the arrow along the tangent of the curve's end, we really move it along the curve by the necessary distance. This operation is more expensive than the `quick` operation – but not *that* expensive, only expensive enough so that it is not selected by default for all arrow tips. Indeed, some compromises are made in the implementation where accuracy was traded for speed, so the distance by which the line end is shifted is not necessarily *exactly* 6.25mm; only something reasonably close.
2. The supports of the line are updated accordingly so that the shortened line will still follow *exactly* the original line. This means that the curve deformation effect caused by the `quick` command does not happen here.
3. Next, the arrow tip is rotated and shifted as follows: First, we shift it so that its tip is exactly at the tip end, where the original line ended. Then, the arrow is rotated so the *the visual end lies on the line*:

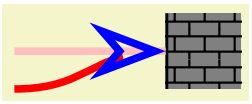


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

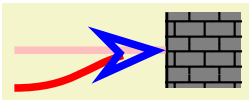
As can be seen in the example, the `flex` option gives a result that is visually pleasing and does not deform the path.

There is, however, one possible problem with the `flex` option: The arrow tip no longer points along the tangent of the end of the path. This may or may not be a problem, put especially for larger arrow tips readers will use the orientation of the arrow head to gauge the direction of the tangent of the line. If this tangent is important (for example, if it should be horizontal), then it may be necessary to enforce that the arrow tip "really points in the direction of the tangent".

To achieve this, the `flex` option takes an optional ⟨*factor*⟩ parameter, which defaults to `1`. This factor specifies how much the arrow tip should be rotated: If set to `0`, the arrow points exactly along a tangent to curve at its tip. If set to `1`, the arrow point exactly along a line from the visual end point on the curve to the tip. For values in the middle, we interpolate the rotation between these two extremes; so `flex=.5` will rotate the arrow's visual end "halfway away from the tangent towards the actual position on the line".

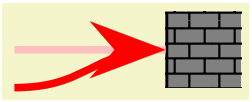


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=0]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



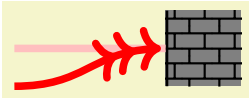
```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex=.5]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Note how in the above examples the red line is visible inside the open arrow tip. Open arrow tips do not go well with a flex value other than `1`. Here is a more realistic use of the `flex=0` key:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,flex=0]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

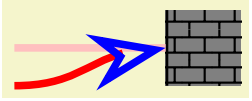
If there are several arrow tips on a path, the `flex` option positions them independently, so that each of them lies optimally on the path:



```
\usetikzlibrary {bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{[flex,sep]>>>}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

`/pgf/arrow keys/flex'`=⟨*factor*⟩ (default `1`)

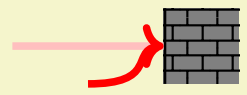
The `flex'` key is almost identical to the `flex` key. The only difference is that a factor of `1` corresponds to rotating the arrow tip so that the instead of the visual end, the "ultimate back end" of the arrow tip lies on the red path. In the example instead of having the arrow tip at a distance of `6.25mm` from the tip lie on the path, we have the point at a distance of `1cm` from the tip lie on the path:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=1cm,open,blue,flex']}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```

Otherwise, the factor works as for `flex` and, indeed `flex=0` and `flex'=0` have the same effect.

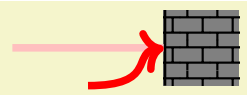
The main use of this option is not so much with an arrow tip like `Stealth` but rather with tips like the standard `>` in the context of a strongly curved line:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Computer Modern Rightarrow[flex]}]
        (0,-.5) .. controls (1,-.5) and (0.5,0) .. (1,0);
\end{tikzpicture}
```

In the example, the `flex` option does not really flex the arrow since for a tip like the Computer Modern arrow, the visual end is the same as the arrow tip – after all, the red line does, indeed, end almost exactly where it used to end.

Nevertheless, you may feel that the arrow tip looks "wrong" in the sense that it should be rotated. This is exactly what the `flex'` option does since it allows us to align the "back end" of the tip with the red line:

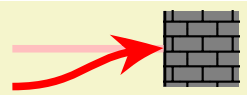


```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Computer Modern Rightarrow[flex'=.75]}]
        (0,-.5) .. controls (1,-.5) and (0.5,0) .. (1,0);
\end{tikzpicture}
```

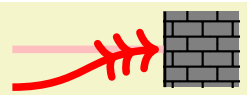
In the example, I used `flex'=.75` so as not to overpronounce the effect. Usually, you will have to fiddle with it sometime to get the "perfectly aligned arrow tip", but a value of `.75` is usually a good start.

`/pgf/arrow keys/bend` (no value)

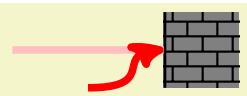
Bending an arrow tip is a radical solution to the problem of positioning arrow tips on a curved line: The arrow tip is no longer "rigid" but the drawing itself will now bend along the curve. This has the advantage that all the problems of flexing with wrong tangents and overflexing disappear. The downsides are longer computation times (bending an arrow is *much* more expensive that flexing it, let alone than quick mode) and also the fact that excessive bending can lead to ugly arrow tips. On the other hand, for most arrow tips their bend version are visually quite pleasing and create a sophisticated look:



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[length=20pt,bend]}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{[bend,sep]>>>}]
        (-1,-.5) .. controls (0,-.5) and (0,0) .. (1,0);
\end{tikzpicture}
```



```
\usetikzlibrary {arrows.meta,bending}
\begin{tikzpicture}
  \wall
  \draw [red!25,line width=1mm] (-1,0) -- (1,0);
  \draw [red,line width=1mm,-{Stealth[bend,round,length=20pt]}]
        (0,-.5) .. controls (1,-.5) and (0.25,0) .. (1,0);
\end{tikzpicture}
```

16.4 Arrow Tip Specifications

16.4.1 Syntax

When you select the arrow tips for the start and the end of a path, you can specify a whole sequence of arrow tips, each having its own local options. At the beginning of this section, it was pointed out that the syntax for selecting the start and end arrow tips is the following:

⟨*start specification*⟩`-`⟨*end specification*⟩

We now have a closer look at what these specifications may look like. The general syntax of the ⟨*start specification*⟩ is as follows:

`[`⟨*options for all tips*⟩`]` ⟨*first arrow tip spec*⟩ ⟨*second arrow tip spec*⟩ ⟨*third arrow tip spec*⟩ ...

As can be seen, an arrow tip specification may start with some options in brackets. If this is the case, the ⟨*options for all tips*⟩ will, indeed, be applied to all arrow tips that follow. (We will see, in a moment, that there are even more places where options may be specified and a list of the ordering in which the options are applied will be given later.)

The main part of a specification is taken up by a sequence of individual arrow tip specifications. Such a specification can be of three kinds:

1. It can be of the form ⟨*arrow tip kind name*⟩`[`⟨*options*⟩`]`.
2. It can be of the form ⟨*shorthand*⟩`[`⟨*options*⟩`]`.
3. It can be of the form ⟨*single char shorthand*⟩`[`⟨*options*⟩`]`. Note that only for this form the brackets are optional.

The easiest kind is the first one: This adds an arrow tip of the kind ⟨*arrow tip kind name*⟩ to the sequence of arrow tips with the ⟨*options*⟩ applied to it at the start (for the ⟨*start specification*⟩) or at the end (for the ⟨*end specification*⟩). Note that for the ⟨*start specification*⟩ the first arrow tip specified in this way will be at the very start of the curve, while for the ⟨*end specification*⟩ the ordering is reversed: The last arrow tip specified will be at the very end of the curve. This implies that a specification like

```
Stealth[] Latex[] - Latex[] Stealth[]
```

will give perfectly symmetric arrow tips on a line (as one would expect).

It is important that even if there are no ⟨*options*⟩ for an arrow tip, the square brackets still need to be written to indicate the end of the arrow tip's name. Indeed, the opening brackets are used to divide the arrow tip specification into names.

Instead of a ⟨*arrow tip kind name*⟩, you may also provide the name of a so-called *shorthand*. Shorthands look like normal arrow tip kind names and, indeed, you will often be using shorthands without noticing that you do. The idea is that instead of, say, `Computer Modern Rightarrow` you might wish to just write `Rightarrow` or perhaps just `To` or even just `>`. For this, you can create a shorthand that tells Ti*k*Z that whenever this shorthand is used, another arrow tip kind is meant. (Actually, shorthands are somewhat more powerful, we have a detailed look at them in Section 16.4.4.) For shorthands, the same rules apply as for normal arrow tip kinds: You *need* to provide brackets so that Ti*k*Z can find the end of the name inside a longer specification.

The third kind of arrow tip specifications consist of just a single letter like `>` or `)` or `*` or even `o` or `x` (but you may not use `[`, `]`, or `-` since they will confuse the parser). These single letter arrow specifications will invariably be shorthands that select some "real" arrow tip instead. An important feature of single letter arrow tips is that they do *not* need to be followed by options (but they may).

Now, since we can use any letter for single letter shorthands, how can Ti*k*Z tell whether by `foo[]` we mean an arrow tip kind `foo` without any options or whether we mean an arrow tip called `f`, followed by two arrow tips called `o`? Or perhaps an arrow tip called `f` followed by an arrow tip called `oo`? To solve this problem, the following rule is used to determine which of the three possible specifications listed above applies: First, we check whether everything from the current position up to the next opening bracket (or up to the end) is the name of an arrow tip or of a shorthand. In our case, `foo` would first be tested under this rule. Only if `foo` is neither the name of an arrow tip kind nor of a shorthand does Ti*k*Z consider the first letter of the specification, `f` in our case. If this is not the name of a shorthand, an error is raised. Otherwise the arrow tip corresponding to `f` is added to the list of arrow tips and the process restarts with the rest.

Thus, we would next text whether `oo` is the name of an arrow tip or shorthand and, if not, whether `o` is such a name.

All of the above rules mean that you can rather easily specify arrow tip sequences if they either mostly consist of single letter names or of longer names. Here are some examples:

- `->>>` is interpreted as three times the `>` shorthand since `>>>` is not the name of any arrow tip kind (and neither is `>>`).
- `->[]>>` has the same effect as the above.
- `-[]>>>` also has the same effect.
- `->[]>[]>[]` so does this.
- `->Stealth` yields an arrow tip `>` followed by a `Stealth` arrow at the end.
- `-Stealth>` is illegal since there is no arrow tip `Stealth>` and since `S` is also not the name of any arrow tip.
- `-Stealth[] >` is legal and does what was presumably meant in the previous item.
- `< Stealth-` is legal and is the counterpart to `-Stealth[] >`.
- `-Stealth[length=5pt] Stealth[length=6pt]` selects two stealth arrow tips, but at slightly different sizes for the end of lines.

An interesting question concerns how flexing and bending interact with multiple arrow tips: After all, flexing and quick mode use different ways of shortening the path so we cannot really mix them. The following rule is used: We check, independently for the start and the end specifications, whether at least one arrow tip in them uses one of the options `flex`, `flex'`, or `bend`. If so, all `quick` settings in the other arrow tips are ignored and treated as if `flex` had been selected for them, too.

16.4.2 Specifying Paddings

When you provide several arrow tips in a row, all of them are added to the start or end of the line:

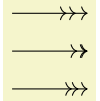

```
\tikz \draw [<<<->>>>] (0,0) -- (2,0);
```

The question now is what will be the distance between them? For this, the following arrow key is important:

`/pgf/arrow keys/sep=`⟨*dimension*⟩ ⟨*line width factor*⟩ ⟨*outer factor*⟩ (default `0.88pt .3 1`)

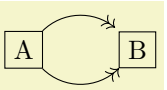
When a sequence of arrow tips is specified in an arrow tip specification for the end of the line, the arrow tips are normally arranged in such a way that the tip of each arrow ends exactly at the "back end" of the next arrow tip (for start specifications, the ordering is inverted, of course). Now, when the `sep` option is set, instead of exactly touching the back end of the next arrow, the specified ⟨*dimension*⟩ is added as additional space (the distance may also be negative, resulting in an overlap of the arrow tips). The optional factors have the same meaning as for the `length` key, see that key for details.

Let us now have a look at some examples. First, we use two arrow tips with different separations between them:



```
\usetikzlibrary {arrows.meta}
\tikz {
  \draw [-{>[sep=1pt]>[sep= 2pt]>}] (0,1.0) -- (1,1.0);
  \draw [-{>[sep=1pt]>[sep=-2pt]>}] (0,0.5) -- (1,0.5);
  \draw [-{>          >[sep]     >}] (0,0.0) -- (1,0.0);
}
```

You can also specify a `sep` for the last arrow tip in the sequence (for end specifications, otherwise for the first arrow tip). In this case, this first arrow tip will not exactly "touch" the point where the path ends, but will rather leave the specified amount of space. This is usually quite desirable.




```
\usetikzlibrary {arrows.meta,positioning}
\tikz {
  \node [draw] (A) {A};
  \node [draw] (B) [right=of A] {B};

  \draw [-{>>[sep=2pt]}] (A) to [bend left=45] (B);
  \draw [- >>          ] (A) to [bend right=45] (B);
}
```

Indeed, adding a `sep` to an arrow tip is *very* desirable, so you will usually write something like `>={To[sep]}` somewhere near the start of your files.

One arrow tip kind can be quite useful in this context: The arrow tip kind `_`. It draws nothing and has zero length, *but* it has `sep` set as a default option. Since it is a single letter shorthand, you can write short and clean "code" in this way:


```
\tikz \draw [->_>] (0,0) -- (1,0);
```

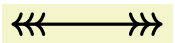


```
\tikz \draw [->__>] (0,0) -- (1,0);
```

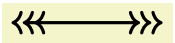
However, using the `sep` option will be faster than using the `_` arrow tip and it also allows you to specify the desired length directly.

16.4.3 Specifying the Line End

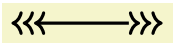
In the previous examples of sequences of arrow tips, the line of the path always ended at the last of the arrow tips (for end specifications) or at the first of the arrow tips (for start specifications). Often, this is what you may want, but not always. Fortunately, it is quite easy to specify the desired end of the line: The special single char shorthand `.` is reserved to indicate that last arrow that is still part of the line; in other words, the line will stop at the last arrow before `.` is encountered (for end specifications) are at the first arrow following `.` (for start specifications).



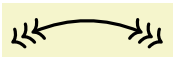
```
\tikz [very thick] \draw [<<<->>>] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<.<<->.>>] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<<.<-.>>>] (0,0) -- (2,0);
```



```
\tikz [very thick] \draw [<<.<->.>>] (0,0) to [bend left] (2,0);
```

It is permissible that there are several dots in a specification, in this case the first one "wins" (for end specifications, otherwise the last one).

Note that `.` is parsed as any other shorthand. In particular, if you wish to add a dot after a normal arrow tip kind name, you need to add brackets:



```
\usetikzlibrary {arrows.meta}
\tikz [very thick] \draw [-{Stealth[] . Stealth[] Stealth[]}] (0,0) -- (2,0);
```

Adding options to `.` is permissible, but they have no effect. In particular, `sep` has no effect since a dot is not an arrow.

16.4.4 Defining Shorthands

It is often desirable to create "shorthands" for the names of arrow tips that you are going to use very often. Indeed, in most documents you will only need a single arrow tip kind and it would be useful that you could refer to it just as `>` in your arrow tip specifications. As another example, you might constantly wish to switch

between a filled and a non-filled circle as arrow tips and would like to use `*` and `o` are shorthands for these case. Finally, you might just like to shorten a long name like `Computer Modern Rightarrow` down to just, say `To` or something similar.

All of these case can be addressed by defining appropriate shorthands. This is done using the following handler:

Key handler ⟨*key*⟩`/.tip`=⟨*end specification*⟩

Defined the ⟨*key*⟩ as a name that can be used inside arrow tip specifications. If the ⟨*key*⟩ has a path before it, this path is ignored (so there is only one "namespace" for arrow tips). Whenever it is used, it will be replaced by the ⟨*end specification*⟩. Note that you must *always* provide (only) an end specification; when the ⟨*key*⟩ is used inside a start specification, the ordering and the meaning of the keys inside the ⟨*end specification*⟩ are translated automatically.


```
\usetikzlibrary {arrows.meta}
\tikz [foo /.tip = {Stealth[sep]. >>}]
  \draw [-foo] (0,0) -- (2,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [foo /.tip = {Stealth[sep] Latex[sep]},
       bar /.tip = {Stealth[length=10pt,open]}]
  \draw [-{foo[red] . bar}] (0,0) -- (2,0);
```

In the last of the examples, we used `foo[red]` to make the arrows red. Any options given to a shorthand upon use will be passed on to the actual arrows tip for which the shorthand stands. Thus, we could also have written `Stealth[sep,red] Latex[sep,red]` instead of `foo[red]`. In other words, the "replacement" of a shorthand by its "meaning" is a semantic replacement rather than a syntactic replacement. In particular, the ⟨*end specification*⟩ will be parsed immediately when the shorthand is being defined. However, this applies only to the options inside the specification, whose values are evaluated immediately. In contrast, which actual arrow tip kind is meant by a given shorthand used inside the ⟨*end specification*⟩ is resolved only up each use of the shorthand. This means that when you write

```
dup /.tip = >>
```

and then later write

```
> /.tip = whatever
```

then `dup` will have the effect as if you had written `whatever[]whatever[]`. You will find that this behavior is what one would expect.

There is one problem we have not yet addressed: The asymmetry of single letter arrow tips like `>` or `)`. When someone writes


```
\tikz \draw [<->] (0,0) -- (1,0);
```

we rightfully expect one arrow tip pointing left at the left end and an arrow tip pointing right at the right end. However, compare


```
\tikz \draw [>->] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz \draw [Stealth-Stealth] (0,0) -- (1,0);
```

In both cases, we have *identical* text in the start and end specifications, but in the first case we rightfully expect the left arrow to be flipped.

The solution to this problem is that it is possible to define two names for the same arrow tip, namely one that is used inside start specifications and one for end specifications. Now, we can decree that the "name of `>`" inside start specifications is simply `<` and the above problems disappear.

To specify different names for a shorthand in start and end specifications, use the following syntax: Instead of ⟨*key*⟩, you use ⟨*name in start specifications*⟩`-`⟨*name in end specifications*⟩. Thus, to set the `>` key correctly, you actually need to write

```
\usetikzlibrary {arrows.meta}
\tikz [<-> /.tip = Stealth] \draw [<->>] (0,0) -- (1,0);
```



```
\usetikzlibrary {arrows.meta}
\tikz [<-> /.tip = Latex] \draw [>-<] (0,0) -- (1,0);
```

Note that the above also works even though we have not set `<` as an arrow tip name for end specifications! The reason this works is that the Ti*k*Z (more precisely, PGF) actually uses the following definition internally:

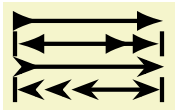
```
>-< /.tip = >[reversed]
```

Translation: "When `<` is used in an end specification, please replace it by `>`, but reversed. Also, when `>` is used in a start specification, we also mean this inverted `>`."

By default, `>` is a shorthand for `To` and `To` is a shorthand for `to` (an arrow from the old libraries) when `arrows.meta` is not loaded library. When `arrows.meta` is loaded, `To` is redefined to mean the same as `Computer Modern Rightarrow`.

`/tikz/>=`⟨*end arrow specification*⟩ (no default)

This is a short way of saying `<->/.tip=`⟨*end arrow specification*⟩.



```
\usetikzlibrary {arrows.meta}
\begin{tikzpicture}[scale=2,ultra thick]
  \begin{scope}[>=Latex]
    \draw[>->]     (0pt,3ex) -- (1cm,3ex);
    \draw[|<->>|] (0pt,2ex) -- (1cm,2ex);
  \end{scope}
  \begin{scope}[>=Stealth]
    \draw[>->]     (0pt,1ex) -- (1cm,1ex);
    \draw[|<<.<->|] (0pt,0ex) -- (1cm,0ex);
  \end{scope}
\end{tikzpicture}
```

`/tikz/shorten <=`⟨*length*⟩ (no default)

Shorten the path by ⟨*length*⟩ in the direction of the starting point.

`/tikz/shorten >=`⟨*length*⟩ (no default)

Shorten the path by ⟨*length*⟩ in the direction of the end point.

16.4.5 Scoping of Arrow Keys

There are numerous places where you can specify keys for an arrow tip. There is, however, one final place that we have not yet mentioned:

`/tikz/arrows=[`⟨*arrow keys*⟩`]` (no default)

The `arrows` key, which is normally used to set the arrow tips for the current scope, can also be used to set some arrow keys for the current scope. When the argument to `arrows` starts with an opening bracket and only otherwise contains one further closing bracket at the very end, this semantic of the `arrow` key is assumed.

The ⟨*arrow keys*⟩ will be set for the rest of current scope. This is useful for generally setting some design parameters or for generally switching on, say, bending as in:

```
\tikz [arrows={[bend]}] ... % Bend all arrows
```

We can now summarize which arrow keys are applied in what order when an arrow tip is used:

1. First, the so-called *defaults* are applied, which are values for the different parameters of a key. They are fixed in the definition of the key and cannot be changed. Since they are executed first, they are only the ultimate fallback.
2. The ⟨*keys*⟩ from the use of `arrows=[`⟨*keys*⟩`]` in all enclosing scopes.

3. Recursively, the ⟨*keys*⟩ provided with the arrow tip inside shorthands.
4. The keys provided at the beginning of an arrow tip specification in brackets.
5. The keys provided directly next to the arrow tip inside the specification.

16.5 Reference: Arrow Tips

TikZ Library arrows.meta

```
\usepgflibrary{arrows.meta}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[arrows.meta]  % ConTeXt when using pure PGF
\usetikzlibrary{arrows.meta} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[arrows.meta] % ConTeXt when using TikZ
```

This library defines a large number of standard "meta" arrow tips. "Meta" means that you can configure these arrow tips in many different ways like changing their size or their line caps and joins and many other details.

The only reason this library is not loaded by default is for compatibility with older versions of TikZ. You can, however, safely load and use this library alongside the older libraries `arrows` and `arrows.spaced`.

The different arrow tip kinds defined in the `arrows.meta` library can be classified in different groups:

- *Barbed* arrow tips consist mainly of lines that "point backward" from the tip of the arrow and which are not filled. For them, filling has no effect. A typical example is ⟶. Here is the list of defined arrow tips:

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Arc Barb[]`	thin	**thick**	
`Bar[]`	thin	**thick**	
`Bracket[]`	thin	**thick**	
`Hooks[]`	thin	**thick**	
`Parenthesis[]`	thin	**thick**	
`Straight Barb[]`	thin	**thick**	
`Tee Barb[]`	thin	**thick**	

All of these arrow tips can be configured and resized in many different ways as described in the following. Above, they are shown at their "natural" sizes, which are chosen in such a way that for a line width of 0.4pt their width matches the height of a letter "x" in Computer Modern at 11pt (with some "overshooting" to create visual consistency).

- *Mathematical* arrow tips are actually a subclass of the barbed arrow tips, but we list them separately. They contain arrow tips that look exactly like the tips of arrows used in mathematical fonts such as the `\to`-symbol → from standard TeX.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Classical TikZ Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[]`	thin	**thick**	
`Implies[]` on double line	thin	**thick**	
`To[]`	thin	**thick**	

The `To` arrow tip is a shorthand for `Computer Modern Rightarrow` when `arrows.meta` is loaded.

- *Geometric* arrow tips consist of a filled shape like a kite or a circle or a "stealth-fighter-like" shape. A typical example is ⟶. These arrow tips can also be used in an "open" variant as in ⟶.

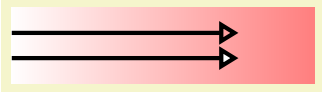
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[]`	thin	**thick**	
`Diamond[]`	thin	**thick**	
`Ellipse[]`	thin	**thick**	
`Kite[]`	thin	**thick**	
`Latex[]`	thin	**thick**	

`Latex[round]`	thin	**thick**	
`Rectangle[]`	thin	**thick**	
`Square[]`	thin	**thick**	
`Stealth[]`	thin	**thick**	
`Stealth[round]`	thin	**thick**	
`Triangle[]`	thin	**thick**	
`Turned Square[]`	thin	**thick**	

Here are the "open" variants:

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[open]`	thin	**thick**	
`Diamond[open]`	thin	**thick**	
`Ellipse[open]`	thin	**thick**	
`Kite[open]`	thin	**thick**	
`Latex[open]`	thin	**thick**	
`Latex[round,open]`	thin	**thick**	
`Rectangle[open]`	thin	**thick**	
`Square[open]`	thin	**thick**	
`Stealth[open]`	thin	**thick**	
`Stealth[round,open]`	thin	**thick**	
`Triangle[open]`	thin	**thick**	
`Turned Square[open]`	thin	**thick**	

Note that "open" arrow tips are not the same as "filled with white", which is also available (just say `fill=white`). The difference is that the background will "shine through" an open arrow, while a filled arrow always obscures the background:



```
\usetikzlibrary {arrows.meta}
\tikz {
  \shade [left color=white, right color=red!50] (0,0) rectangle (4,1);

  \draw [ultra thick,-{Triangle[open]}]       (0,2/3) -- ++ (3,0);
  \draw [ultra thick,-{Triangle[fill=white]}] (0,1/3) -- ++ (3,0);
}
```

- *Cap* arrow tips are used to add a "cap" to the end of a line. The graphic languages underlying TikZ (PDF, POSTSCRIPT or SVG) all support three basic types of line caps on a very low level: round, rectangular, and "butt". Using cap arrow tips, you can add new caps to lines and use different caps for the end and the start. An example is the line ➠»».

Appearance of the below at line width	*1ex*	*1em*
`Butt Cap[]`		
`Fast Round[]`		
`Fast Triangle[]`		
`Round Cap[]`		
`Triangle Cap[]`		

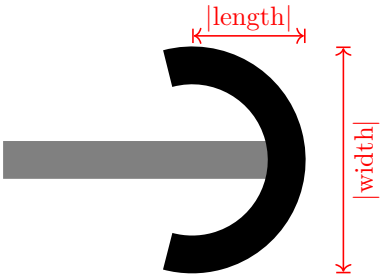
- *Special* arrow tips are used for some specific purpose and do not fit into the above categories.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Rays[]`	thin	**thick**	
`Rays[n=8]`	thin	**thick**	

16.5.1 Barbed Arrow Tips

Arrow Tip Kind `Arc Barb`

This arrow tip attaches an arc to the end of the line whose angle is given by the `arc` option. The `length` and `width` parameters refer to the size of the arrow tip for `arc` set to 180 degrees, which is why in the example for `arc=210` the actual length is larger than the specified `length`. The line width is taken into account for the computation of the length and width. Use the `round` option to add round caps to the end of the arcs.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Arc Barb[]`	thin	**thick**	
`Arc Barb[sep] Arc Barb[]`	thin	**thick**	
`Arc Barb[sep] . Arc Barb[]`	thin	**thick**	
`Arc Barb[arc=120]`	thin	**thick**	
`Arc Barb[arc=270]`	thin	**thick**	
`Arc Barb[length=2pt]`	thin	**thick**	
`Arc Barb[length=2pt,width=5pt]`	thin	**thick**	
`Arc Barb[line width=2pt]`	thin	**thick**	
`Arc Barb[reversed]`	thin	**thick**	
`Arc Barb[round]`	thin	**thick**	
`Arc Barb[slant=.3]`	thin	**thick**	
`Arc Barb[left]`	thin	**thick**	
`Arc Barb[right]`	thin	**thick**	
`Arc Barb[harpoon,reversed]`	thin	**thick**	
`Arc Barb[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Bar`

A simple bar. This is a simple instance of `Tee Barb` for length zero.

Arrow Tip Kind `Bracket`

This is an instance of the `Tee Barb` arrow tip that results in something resembling a bracket. Just like the `Parenthesis` arrow tip, a `Bracket` is not modelled from a text square bracket, but rather its size has been chosen so that it fits with the other arrow tips.



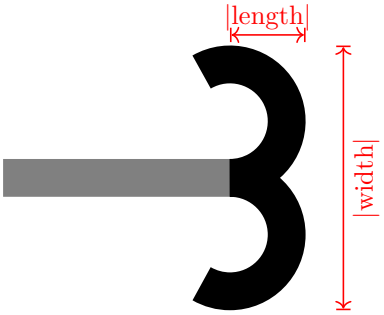
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Bracket[]`	thin	**thick**	
`Bracket[sep] Bracket[]`	thin	**thick**	
`Bracket[sep] . Bracket[]`	thin	**thick**	
`Bracket[reversed]`	thin	**thick**	
`Bracket[round]`	thin	**thick**	
`Bracket[slant=.3]`	thin	**thick**	
`Bracket[left]`	thin	**thick**	
`Bracket[right]`	thin	**thick**	
`Bracket[harpoon,reversed]`	thin	**thick**	
`Bracket[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Hooks`

This arrow tip attaches two “hooks” to the end of the line. The `length` and `width` parameters refer to the size of the arrow tip if both arcs are 180 degrees; in the example the arc is 210 degrees and, thus, the arrow is actually longer that the `length` dictates. The line width is taken into account for the computation of the length and width. The `arc` option is used to specify the angle of the arcs. Use the `round` option to add round caps to the end of the arcs.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Hooks[]`	thin	**thick**	
`Hooks[sep] Hooks[]`	thin	**thick**	
`Hooks[sep] . Hooks[]`	thin	**thick**	
`Hooks[arc=120]`	thin	**thick**	
`Hooks[arc=270]`	thin	**thick**	
`Hooks[length=2pt]`	thin	**thick**	
`Hooks[length=2pt,width=5pt]`	thin	**thick**	
`Hooks[line width=2pt]`	thin	**thick**	
`Hooks[reversed]`	thin	**thick**	
`Hooks[round]`	thin	**thick**	
`Hooks[slant=.3]`	thin	**thick**	
`Hooks[left]`	thin	**thick**	
`Hooks[right]`	thin	**thick**	
`Hooks[harpoon,reversed]`	thin	**thick**	
`Hooks[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Parenthesis`

This arrow tip is an instantiation of the `Arc Barb` so that it resembles a parenthesis. However, the idea is not to recreate a “real” parenthesis as it is used in text, but rather a “bow” at a size that harmonizes with the other arrow tips at their default sizes.



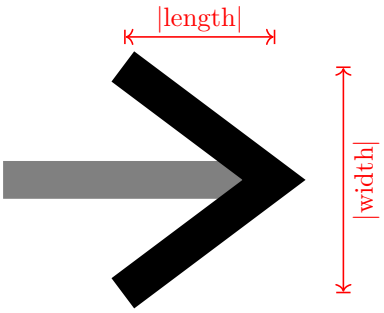
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Parenthesis[]`	thin	**thick**	
`Parenthesis[sep] Parenthesis[]`	thin	**thick**	
`Parenthesis[sep] . Parenthesis[]`	thin	**thick**	
`Parenthesis[reversed]`	thin	**thick**	
`Parenthesis[round]`	thin	**thick**	
`Parenthesis[slant=.3]`	thin	**thick**	
`Parenthesis[left]`	thin	**thick**	
`Parenthesis[right]`	thin	**thick**	
`Parenthesis[harpoon,reversed]`	thin	**thick**	
`Parenthesis[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Straight Barb`

This is the “archetypal” arrow head, consisting of just two straight lines. The `length` and `width` parameters refer to the horizontal and vertical distances between the points on the path making up the arrow tip. As can be seen, the line width of the arrow tip's path is not taken into account. The `angle` option is particularly useful to set the opening angle at the tip of the arrow head. The `round` option gives a “softer” or “rounder” version of the arrow tip.



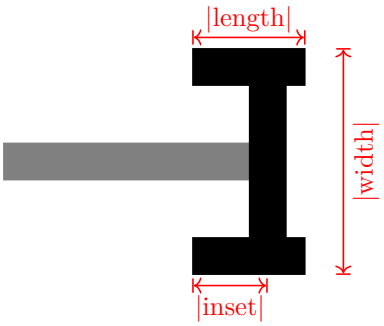
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Straight Barb[]`	thin	**thick**	
`Straight Barb[] Straight Barb[]`	thin	**thick**	
`Straight Barb[] . Straight Barb[]`	thin	**thick**	
`Straight Barb[length=5pt]`	thin	**thick**	
`Straight Barb[length=5pt,width=5pt]`	thin	**thick**	
`Straight Barb[line width=2pt]`	thin	**thick**	
`Straight Barb[reversed]`	thin	**thick**	
`Straight Barb[angle=60:2pt 3]`	thin	**thick**	
`Straight Barb[round]`	thin	**thick**	
`Straight Barb[slant=.3]`	thin	**thick**	
`Straight Barb[left]`	thin	**thick**	
`Straight Barb[right]`	thin	**thick**	
`Straight Barb[harpoon,reversed]`	thin	**thick**	
`Straight Barb[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Tee Barb`

This arrow tip attaches a little “T” on both sides of the tip. The arrow `inset` dictates the distance from the back end to the middle of the stem of the T. When the inset is equal to the length, the arrow tip is drawn as a single line, not as three lines (this is important for the “round” version since, then, the corners get rounded).



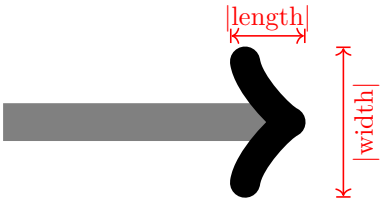
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Tee Barb[]`	thin	**thick**	
`Tee Barb[sep] Tee Barb[]`	thin	**thick**	
`Tee Barb[sep] . Tee Barb[]`	thin	**thick**	
`Tee Barb[inset=0pt]`	thin	**thick**	
`Tee Barb[inset'=0pt 1]`	thin	**thick**	
`Tee Barb[line width=2pt]`	thin	**thick**	
`Tee Barb[round]`	thin	**thick**	
`Tee Barb[round,inset'=0pt 1]`	thin	**thick**	
`Tee Barb[slant=.3]`	thin	**thick**	
`Tee Barb[left]`	thin	**thick**	
`Tee Barb[right]`	thin	**thick**	
`Tee Barb[harpoon,reversed]`	thin	**thick**	
`Tee Barb[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

16.5.2 Mathematical Barbed Arrow Tips

Arrow Tip Kind `Classical TikZ Rightarrow`



This arrow tip is the "old" or "classical" arrow tip that used to be the standard in Ti*k*Z in earlier versions. It was modelled on an old version of the tip of `\rightarrow` (→) of the Computer Modern fonts. However, this "old version" was really old, Donald Knuth (the designer of both TEX and of the Computer Modern fonts) replaced the arrow tip of the mathematical fonts in 1992.

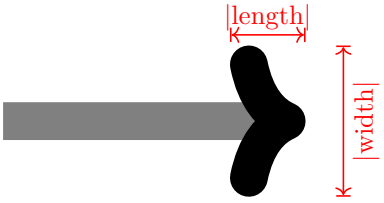
The main problem with this arrow tip is that it is "too small" at its natural size. I recommend using the new `Computer Modern Rightarrow` arrow tip instead, which matches the current →. This new version is also the default used as `>` and as `To`, now.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[sep] Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[sep] . Classical TikZ Rightarrow[]`	thin	**thick**	
`Classical TikZ Rightarrow[length=3pt]`	thin	**thick**	
`Classical TikZ Rightarrow[sharp]`	thin	**thick**	
`Classical TikZ Rightarrow[slant=.3]`	thin	**thick**	
`Classical TikZ Rightarrow[left]`	thin	**thick**	
`Classical TikZ Rightarrow[right]`	thin	**thick**	
`Classical TikZ Rightarrow[harpoon,reversed]`	thin	**thick**	
`Classical TikZ Rightarrow[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Computer Modern Rightarrow`



For a line width of 0.4pt (the default), this arrow tip looks very much like `\rightarrow` (→) of the Computer Modern math fonts. However, it is not a "perfect" match: the line caps and joins of the "real" → are rounded differently from this arrow tip; but it takes a keen eye to notice the difference. When the `arrows.meta` library is loaded, this arrow tip becomes the default of `To` and, thus, is used whenever `>` is used (unless, of course, you redefined `>`).

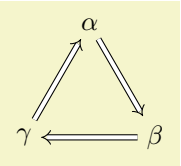
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[sep] Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[sep] . Computer Modern Rightarrow[]`	thin	**thick**	
`Computer Modern Rightarrow[length=3pt]`	thin	**thick**	
`Computer Modern Rightarrow[sharp]`	thin	**thick**	
`Computer Modern Rightarrow[slant=.3]`	thin	**thick**	
`Computer Modern Rightarrow[left]`	thin	**thick**	
`Computer Modern Rightarrow[right]`	thin	**thick**	
`Computer Modern Rightarrow[harpoon,reversed]`	thin	**thick**	
`Computer Modern Rightarrow[red]`	thin	**thick**	

The following options have no effect: `open`, `fill`.

On `double` lines, the arrow tip will not look correct.

Arrow Tip Kind `Implies`

This arrow tip makes only sense in conjunction with the `double` option. The idea is that you attach it to a double line to get something that looks like TEX's `\implies` arrow (⟹). A typical use of this arrow tip is




```
\usetikzlibrary {arrows.meta,graphs}
\tikz \graph [clockwise=3, math nodes,
              edges = {double equal sign distance, -Implies}] {
  "\alpha", "\beta", "\gamma";
  "\alpha" -> "\beta" -> "\gamma" -> "\alpha"
};
```

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Implies[]` on double line	thin	**thick**	
`Implies[red]` on double line	thin	**thick**	

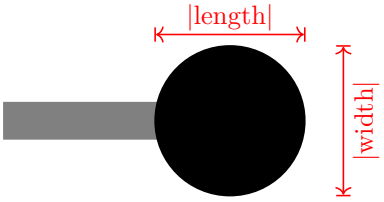
Arrow Tip Kind `To`

This is a shorthand for `Computer Modern Rightarrow` when the `arrows.meta` library is loaded. Otherwise, it is a shorthand for the classical Ti*k*Z rightarrow.

16.5.3 Geometric Arrow Tips

Arrow Tip Kind `Circle`

Although this tip is called "circle", you can also use it to draw ellipses if you set the length and width to different values. Neither `round` nor `reversed` has any effect on this arrow tip.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Circle[]`	thin	**thick**	
`Circle[sep] Circle[]`	thin	**thick**	
`Circle[sep] . Circle[]`	thin	**thick**	
`Circle[open]`	thin	**thick**	
`Circle[length=3pt]`	thin	**thick**	
`Circle[slant=.3]`	thin	**thick**	
`Circle[left]`	thin	**thick**	
`Circle[right]`	thin	**thick**	
`Circle[red]`	thin	**thick**	

Arrow Tip Kind `Diamond`

This is an instance of `Kite` where the length is larger than the width.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Diamond[]`	thin	**thick**	
`Diamond[] Diamond[]`	thin	**thick**	
`Diamond[] . Diamond[]`	thin	**thick**	
`Diamond[open]`	thin	**thick**	
`Diamond[length=10pt]`	thin	**thick**	
`Diamond[round]`	thin	**thick**	
`Diamond[slant=.3]`	thin	**thick**	
`Diamond[left]`	thin	**thick**	
`Diamond[right]`	thin	**thick**	
`Diamond[red]`	thin	**thick**	
`Diamond[fill=red!50]`	thin	**thick**	

Arrow Tip Kind `Ellipse`

This is a shorthand for a "circle" that is twice as wide as high.

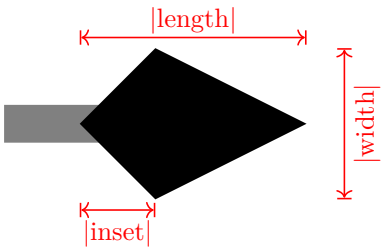
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Ellipse[]`	thin	**thick**	

```
Ellipse[sep] Ellipse[]
Ellipse[sep] . Ellipse[]
Ellipse[open]
Ellipse[length=10pt]
Ellipse[round]
Ellipse[slant=.3]
Ellipse[left]
Ellipse[right]
Ellipse[red]
Ellipse[fill=red!50]
```

thin **thick**

Arrow Tip Kind `Kite`

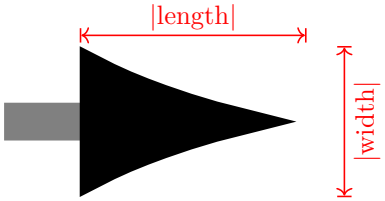
This arrow tip consists of four lines that form a "kite". The `inset` prescribed how far the width-axis of the kite is removed from the back end. Note that the inset cannot be negative, use a `Stealth` arrow tip for this.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Kite[]`	thin	**thick**	
`Kite[sep] Kite[]`	thin	**thick**	
`Kite[sep] . Kite[]`	thin	**thick**	
`Kite[open]`	thin	**thick**	
`Kite[length=6pt,width=4pt]`	thin	**thick**	
`Kite[length=6pt,width=4pt,inset=1.5pt]`	thin	**thick**	
`Kite[round]`	thin	**thick**	
`Kite[slant=.3]`	thin	**thick**	
`Kite[left]`	thin	**thick**	
`Kite[right]`	thin	**thick**	
`Kite[red]`	thin	**thick**	

Arrow Tip Kind `Latex`

This arrow tip is the same as the arrow tip used in LaTeX's standard pictures (via the `\vec` command), if you set the length to 4pt. The default size for this arrow tip was set slightly larger so that it fits better with the other geometric arrow tips.



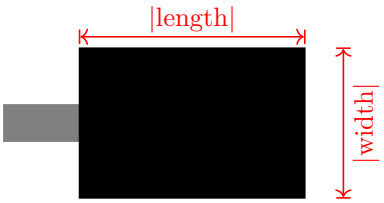
Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Latex[]`	thin	**thick**	
`Latex[sep] Latex[]`	thin	**thick**	
`Latex[sep] . Latex[]`	thin	**thick**	
`Latex[open]`	thin	**thick**	
`Latex[length=4pt]`	thin	**thick**	
`Latex[round]`	thin	**thick**	
`Latex[slant=.3]`	thin	**thick**	
`Latex[left]`	thin	**thick**	
`Latex[right]`	thin	**thick**	
`Latex[red]`	thin	**thick**	

Arrow Tip Kind `LaTeX`

Another spelling for the `Latex` arrow tip.

Arrow Tip Kind `Rectangle`

A rectangular arrow tip. By default, it is twice as long as high.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Rectangle[]`	thin	**thick**	
`Rectangle[sep] Rectangle[]`	thin	**thick**	
`Rectangle[sep] . Rectangle[]`	thin	**thick**	
`Rectangle[open]`	thin	**thick**	
`Rectangle[length=4pt]`	thin	**thick**	
`Rectangle[round]`	thin	**thick**	
`Rectangle[slant=.3]`	thin	**thick**	
`Rectangle[left]`	thin	**thick**	
`Rectangle[right]`	thin	**thick**	
`Rectangle[red]`	thin	**thick**	

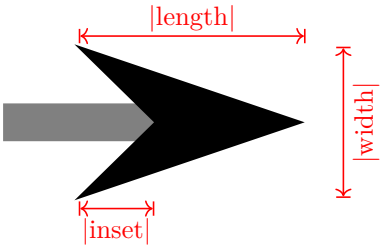
Arrow Tip Kind `Square`

An instance of the `Rectangle` whose width is identical to the length.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Square[]`	thin	**thick**	
`Square[sep] Square[]`	thin	**thick**	
`Square[sep] . Square[]`	thin	**thick**	
`Square[open]`	thin	**thick**	
`Square[length=4pt]`	thin	**thick**	
`Square[round]`	thin	**thick**	
`Square[slant=.3]`	thin	**thick**	
`Square[left]`	thin	**thick**	
`Square[right]`	thin	**thick**	
`Square[red]`	thin	**thick**	

Arrow Tip Kind `Stealth`

This arrow tip is similar to a `Kite`, only the `inset` now counts "inwards". Because of that sharp angles, for this arrow tip is makes quite a difference, visually, if use the `round` option. Also, using the `harpoon` option (or `left` or `right`) will *lengthen* the arrow tip because of the even sharper corner at the tip.



Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Stealth[]`	thin	**thick**	
`Stealth[sep] Stealth[]`	thin	**thick**	
`Stealth[sep] . Stealth[]`	thin	**thick**	
`Stealth[open]`	thin	**thick**	
`Stealth[length=6pt,width=4pt]`	thin	**thick**	
`Stealth[length=6pt,width=4pt,inset=1.5pt]`	thin	**thick**	
`Stealth[round]`	thin	**thick**	
`Stealth[slant=.3]`	thin	**thick**	
`Stealth[left]`	thin	**thick**	
`Stealth[right]`	thin	**thick**	
`Stealth[red]`	thin	**thick**	

Arrow Tip Kind `Triangle`

An instance of a `Kite` with zero inset.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Triangle[]`	thin	**thick**	
`Triangle[sep] Triangle[]`	thin	**thick**	
`Triangle[sep] . Triangle[]`	thin	**thick**	
`Triangle[open]`	thin	**thick**	
`Triangle[length=4pt]`	thin	**thick**	
`Triangle[angle=45:1pt 3]`	thin	**thick**	
`Triangle[angle=60:1pt 3]`	thin	**thick**	
`Triangle[angle=90:1pt 3]`	thin	**thick**	
`Triangle[round]`	thin	**thick**	
`Triangle[slant=.3]`	thin	**thick**	
`Triangle[left]`	thin	**thick**	
`Triangle[right]`	thin	**thick**	
`Triangle[red]`	thin	**thick**	

Arrow Tip Kind `Turned Square`

An instance of a `Kite` with identical width and height and mid-inset.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Turned Square[]`	thin	**thick**	
`Turned Square[sep] Turned Square[]`	thin	**thick**	
`Turned Square[sep] . Turned Square[]`	thin	**thick**	
`Turned Square[open]`	thin	**thick**	
`Turned Square[length=4pt]`	thin	**thick**	
`Turned Square[round]`	thin	**thick**	
`Turned Square[slant=.3]`	thin	**thick**	
`Turned Square[left]`	thin	**thick**	
`Turned Square[right]`	thin	**thick**	
`Turned Square[red]`	thin	**thick**	

16.5.4 Caps

Recall that a *cap* is a way of ending a line. The graphic languages underlying Ti*k*Z (PDF, POSTSCRIPT or SVG) all support three basic types of line caps on a very low level: round, rectangular, and "butt". Using cap arrow tips, you can add new caps to lines and use different caps for the end and the start.

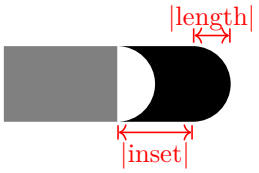
Arrow Tip Kind `Butt Cap`

This arrow tip ends the line "in the normal way" with a straight end. This arrow tip is only need to "cover up" the actual line cap, if this happens to differ from the normal cap. In the following example, the line cap is "round", but, nevertheless, the right end is a "butt" cap:


```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex, line cap=round, -Butt Cap] (0,0) -- (1,0);
```

Arrow Tip Kind `Fast Round`

This arrow tip is not really a cap, you use it in conjunction with (typically) the `Round Cap`. The idea is that you end your line using the round cap and then add several `Fast Round`s. As for `Round Cap`, the `length` parameter dictates the length is the length of the "main part", the inset sets the length of a line that comes before this tip.




```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex,
             -{Round Cap [] . Fast Round[] Fast Round[]}]
  (0,0) -- (1,0);
```

Note that in conjunction with the `bend` option, this works even quite well for curves:

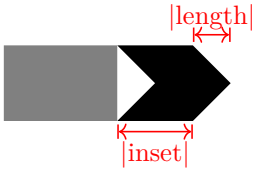


```
\usetikzlibrary {arrows.meta,bending}
\tikz [f/.tip = Fast Round] % shorthand
  \draw [line width=1ex, -{[bend] Round Cap[] . f f f}]
  (0,0) to [bend left] (1,0);
```

Appearance of the below at line width	*1ex*	*1em*
`Fast Round[]`		
`Fast Round[reversed]`		
`Fast Round[cap angle=60]`		
`Fast Round[cap angle=60,inset=5pt]`		
`Fast Round[length=.5ex]`		
`Fast Round[slant=.3]`		

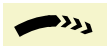
Arrow Tip Kind `Fast Triangle`

This arrow tip works like `Fast Round`, only for triangular caps.




```
\usetikzlibrary {arrows.meta}
\tikz \draw [line width=1ex,
             -{Triangle Cap []. Fast Triangle[] Fast Triangle[]}]
  (0,0) -- (1,0);
```

Again, this tip works well for curves:



```
\usetikzlibrary {arrows.meta,bending}
\tikz [f/.tip = Fast Triangle] % shorthand
  \draw [line width=1ex, -{[bend] Triangle Cap[] . f f f}]
  (0,0) to [bend left] (1,0);
```

Appearance of the below at line width	*1ex*	*1em*
`Fast Triangle[]`		
`Fast Triangle[reversed]`		
`Fast Triangle[cap angle=60]`		
`Fast Triangle[cap angle=60,inset=5pt]`		
`Fast Triangle[length=.5ex]`		
`Fast Triangle[slant=.3]`		

Arrow Tip Kind `Round Cap`

This arrow tip ends the line using a half circle or, if the length has been modified, a half-ellipse.



Appearance of the below at line width	*1ex*	*1em*
`Round Cap[]`		
`Round Cap[reversed]`		
`Round Cap[length=.5ex]`		
`Round Cap[slant=.3]`		

Arrow Tip Kind `Triangle Cap`

This arrow tip ends the line using a triangle whose length is given by the `length` option.



You can get any angle you want at the tip by specifying a length that is an appropriate multiple of the line width. The following options does this computation for you:

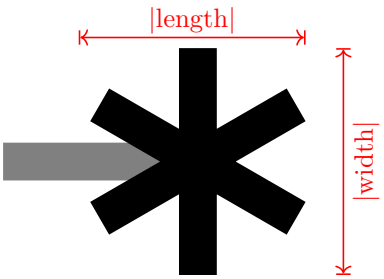
/pgf/arrow keys/cap angle=⟨*angle*⟩ (no default)

Sets `length` to an appropriate multiple of the line width so that the angle of a `Triangle Cap` is exactly ⟨*angle*⟩ at the tip.

Appearance of the below at line width	*1ex*	*1em*
`Triangle Cap[]`		
`Triangle Cap[reversed]`		
`Triangle Cap[cap angle=60]`		
`Triangle Cap[cap angle=60,reversed]`		
`Triangle Cap[length=.5ex]`		
`Triangle Cap[slant=.3]`		

16.5.5 Special Arrow Tips

Arrow Tip Kind `Rays`



This arrow tip attaches a "bundle of rays" to the tip. The number of evenly spaced rays is given by the `n` arrow key (see below). When the number is even, the rays will lie to the left and to the right of the direction of the arrow; when the number is odd, the rays are rotated in such a way that one of them points perpendicular to the direction of the arrow (this is to ensure that no ray points in the direction of the line, which would look strange). The `length` and `width` describe the length and width of an ellipse into which the rays fit.

Appearance of the below at line width	*0.4pt*	*0.8pt*	*1.6pt*
`Rays[]`	thin	**thick**	
`Rays[sep] Rays[]`	thin	**thick**	
`Rays[sep] . Rays[]`	thin	**thick**	
`Rays[width'=0pt 2]`	thin	**thick**	
`Rays[round]`	thin	**thick**	
`Rays[n=2]`	thin	**thick**	
`Rays[n=3]`	thin	**thick**	
`Rays[n=4]`	thin	**thick**	
`Rays[n=5]`	thin	**thick**	
`Rays[n=6]`	thin	**thick**	
`Rays[n=7]`	thin	**thick**	
`Rays[n=8]`	thin	**thick**	
`Rays[n=9]`	thin	**thick**	
`Rays[slant=.3]`	thin	**thick**	
`Rays[left]`	thin	**thick**	
`Rays[right]`	thin	**thick**	
`Rays[left,n=5]`	thin	**thick**	
`Rays[right,n=5]`	thin	**thick**	
`Rays[red]`	thin	**thick**	

/pgf/arrow keys/n=⟨*number*⟩ (no default, initially 4)

Sets the number of rays in a `Rays` arrow tip.

17 Nodes and Edges

17.1 Overview

In the present section, the usage of *nodes* in Ti*k*Z is explained. A node is typically a rectangle or circle or another simple shape with some text on it.

Nodes are added to paths using the special path operation `node`. Nodes *are not part of the path itself.* Rather, they are added to the picture just before or after the path has been drawn.

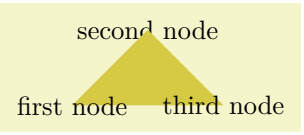
In Section 17.2 the basic syntax of the node operation is explained, followed in Section 17.3 by the syntax for multi-part nodes, which are nodes that contain several different text parts. After this, the different options for the text in nodes are explained. In Section 17.5 the concept of *anchors* is introduced along with their usage. In Section 17.7 the different ways transformations affect nodes are studied. Sections 17.8 and 17.9 are about placing nodes on or next to straight lines and curves. Section 17.11 explains how a node can be used as a "pseudo-coordinate". Section 17.12 introduces the `edge` operation, which works similar to the `to` operation and also similar to the `node` operation.

17.2 Nodes and Their Shapes

In the simplest case, a node is just some text that is placed at some coordinate. However, a node can also have a border drawn around it or have a more complex background and foreground. Indeed, some nodes do not have a text at all, but consist solely of the background. You can name nodes so that you can reference their coordinates later in the same picture or, if certain precautions are taken as explained in Section 17.13, also in different pictures.

There are no special TEX commands for adding a node to a picture; rather, there is path operation called `node` for this. Nodes are created whenever Ti*k*Z encounters `node` or `coordinate` at a point on a path where it would expect a normal path operation (like `-- (1,1)` or `rectangle (1,1)`). It is also possible to give node specifications *inside* certain path operations as explained later.

The node operation is typically followed by some options, which apply only to the node. Then, you can optionally *name* the node by providing a name in parentheses. Lastly, for the `node` operation you must provide some label text for the node in curly braces, while for the `coordinate` operation you may not. The node is placed at the current position of the path either *after the path has been drawn* or (more seldom and only if you add the `behind path` option) *just before the path is drawn.* Thus, all nodes are drawn "on top" or "behind" the path and are retained until the path is complete. If there are several nodes on a path, perhaps some behind and some on top of the path, first come the nodes behind the path in the order they were encountered, then comes that path, and then come the remaining node, again in the order they are encountered.




```
\tikz \fill [fill=yellow!80!black]
     (0,0) node              {first node}
  -- (1,1) node[behind path] {second node}
  -- (2,0) node              {third node};
```

17.2.1 Syntax of the Node Command

The syntax for specifying nodes is the following:

`\path` … `node` ⟨*foreach statements*⟩ `[`⟨*options*⟩`]` `(`⟨*name*⟩`)` `at(`⟨*coordinate*⟩`)` `:`⟨*animation attribute*⟩`={`⟨*options*⟩`}` `{`⟨*node contents*⟩`}` … `;`

Since this path operation is one of the most involved around, let us go over it step by step.

Order of the parts of the specification. Everything between "`node`" and the opening brace of a node is optional. If there are ⟨*foreach statements*⟩, they must come first, directly following "`node`". Other than that, the ordering of all the other elements of a node specification (the ⟨*options*⟩, the ⟨*name*⟩, ⟨*coordinate*⟩, and ⟨*animation attribute*⟩) is arbitrary, indeed, there can be multiple occurrences of any of these elements (although for the name and the coordinate this makes no sense).

The text of a node. At the end of a node, you must (normally) provide some ⟨*node contents*⟩ in curly braces; indeed, the "end" of the node specification is detected by the opening curly brace. For normal nodes it is possible to use "fragile" stuff inside the ⟨*node contents*⟩ like the `\verb` command (for the

technically savvy: code inside the ⟨*node contents*⟩ is allowed to change catcodes; however, this rule does not apply to "nodes on a path" to be discussed later).

Instead of giving ⟨*node contents*⟩ at the end of the node in curly braces, you can also use the following key:

`/tikz/node contents=`⟨*node contents*⟩ (no default)

This key sets the contents of the node to the given text as if you had given it at the end in curly braces. When the option is used inside the options of a node, the parsing of the node stops immediately after the end of the option block. In particular, the option block cannot be followed by further option blocks or curly braces (or, rather, these do not count as part of the node specification.) Also note that the ⟨*node contents*⟩ may not contain fragile stuff since the catcodes get fixed upon reading the options. Here is an example:




```
\tikz {
  \path (0,0) node [red]                    {A}
        (1,0) node [blue]                   {B}
        (2,0) node [green, node contents=C]
        (3,0) node [node contents=D]           ;
}
```

Specifying the location of the node. Nodes are placed at the last position mentioned on the path. The effect of adding "`at`" to a node specification is that the coordinate given after `at` is used instead. The `at` syntax is not available when a node is given inside a path operation (it would not make any sense there).

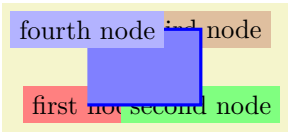
`/tikz/at=`⟨*coordinate*⟩ (no default)

This is another way of specifying the `at` coordinate. Note that, typically, you will have to enclose the ⟨*coordinate*⟩ in curly braces so that a comma inside the ⟨*coordinate*⟩ does not confuse TEX.

Another aspect of the "location" of a node is whether it appears *in front of* or *behind* the current path. You can change which of these two possibilities happens on a node-by-node basis using the following keys:

`/tikz/behind path` (no value)

When this key is set, either as a local option for the node or some surrounding scope, the node will be drawn behind the current path. For this, TikZ collects all nodes defined on the current path with this option set and then inserts all of them, in the order they appear, just before it draws the path. Thus, several nodes with this option set may obscure one another, but never the path itself. "Just before it draws the path" actually means that the nodes are inserted into the page output just before any pre-actions are applied to the path (see below for what pre-actions are).




```
\tikz \fill [fill=blue!50, draw=blue, very thick]
      (0,0)   node [behind path, fill=red!50]   {first node}
   -- (1.5,0) node [behind path, fill=green!50] {second node}
   -- (1.5,1) node [behind path, fill=brown!50] {third node}
   -- (0,1)   node [             fill=blue!30]  {fourth node};
```

Note that `behind path` only applies to the current path; not to the current scope or picture. To put a node "behind everything" you need to use layers and options like `on background layer`, see the `backgrounds` library in Section 45.

`/tikz/in front of path` (no value)

This is the opposite of `behind path`: It causes nodes to be drawn on top of the path. Since this is the default behavior, you usually do not need this option; it is only needed when an enclosing scope has used `behind path` and you now wish to "switch back" to the normal behavior.

The name of a node. The `(`⟨*name*⟩`)` is a name for later reference and it is optional. You may also add the option `name=`⟨*name*⟩ to the ⟨*option*⟩ list; it has the same effect.

/tikz/name=⟨*node name*⟩ (no default)

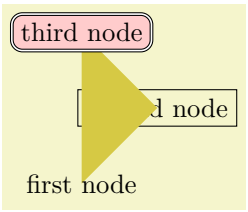
Assigns a name to the node for later reference. Since this is a "high-level" name (drivers never know of it), you can use spaces, number, letters, or whatever you like when naming a node. Thus, you can name a node just `1` or perhaps `start of chart` or even `y_1`. Your node name should *not* contain any punctuation like a dot, a comma, or a colon since these are used to detect what kind of coordinate you mean when you reference a node.

/tikz/alias=⟨*another node name*⟩ (no default)

This option allows you to provide another name for the node. Giving this option multiple times will allow you to access the node via several aliases. Using the `node also` syntax, you can also assign an alias name to a node at a later point, see Section 17.14.

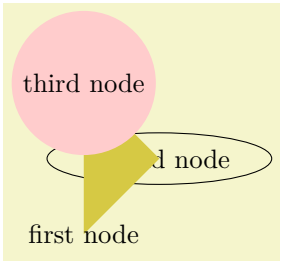
The options of a node. The ⟨*options*⟩ is an optional list of options that *apply only to the node* and have no effect outside. The other way round, most "outside" options also apply to the node, but not all. For example, the "outside" rotation does not apply to nodes (unless some special options are used, sigh). Also, the outside path action, like `draw` or `fill`, never applies to the node and must be given in the node (unless some special other options are used, deep sigh).

The shape of a node. As mentioned before, we can add a border and even a background to a node:




```
\tikz \fill[fill=yellow!80!black]
      (0,0) node {first node}
   -- (1,1) node[draw, behind path] {second node}
   -- (0,2) node[fill=red!20,draw,double,rounded corners] {third node};
```

The "border" is actually just a special case of a much more general mechanism. Each node has a certain *shape* which, by default, is a rectangle. However, we can also ask Ti*k*Z to use a circle shape instead or an ellipse shape (you have to include one of the `shapes.geometric` library for the latter shape):




```
\usetikzlibrary {shapes.geometric}
\tikz \fill[fill=yellow!80!black]
      (0,0) node                                {first node}
   -- (1,1) node[ellipse,draw, behind path] {second node}
   -- (0,2) node[circle,fill=red!20]           {third node};
```

There are many more shapes available such as, say, a shape for a resistor or a large arrow, see the `shapes` library in Section 71 for details.

To select the shape of a node, the following option is used:

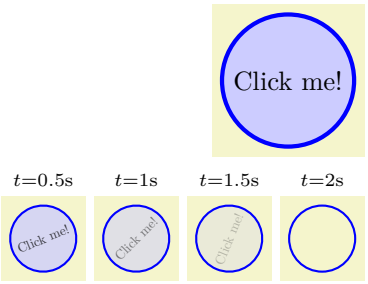
/tikz/shape=⟨*shape name*⟩ (no default, initially `rectangle`)

Select the shape either of the current node or, when this option is not given inside a node but somewhere outside, the shape of all nodes in the current scope.

Since this option is used often, you can leave out the `shape=`. When Ti*k*Z encounters an option like `circle` that it does not know, it will, after everything else has failed, check whether this option is the name of some shape. If so, that shape is selected as if you had said `shape=`⟨*shape name*⟩.

By default, the following shapes are available: `rectangle`, `circle`, `coordinate`. Details of these shapes, like their anchors and size options, are discussed in Section 17.2.2.

Animating a node. When you say `:`⟨*animation attribute*⟩`={`⟨*options*⟩`}`, an *animation* of the specified attribute is added to the node. Animations are discussed in detail in Section 26. Here is a typical example of how this syntax can be used:




```
\usetikzlibrary {animations}
\tikz
  \node  :fill opacity = { 0s="1", 2s="0", begin on=click }
         :rotate = { 0s="0", 2s="90", begin on=click }
         [fill = blue!20, draw = blue, ultra thick, circle]
    {Click me!};
```

The foreach statement for nodes. At the beginning of a node specification (and only there) you can provide multiple ⟨*foreach statements*⟩, each of which has the form `foreach` ⟨*var*⟩ `in` ⟨*list*⟩ (note that there is no backslash before `foreach`). When they are given, instead of a single node, multiple nodes will be created: The ⟨*var*⟩ will iterate over all values of ⟨*list*⟩ and for each of them, a new node is created. These nodes are all created using all the text following the ⟨*foreach statements*⟩, but in each copy the ⟨*var*⟩ will have the current value of the current element in the ⟨*list*⟩.

As an example, the following two codes have the same effect:

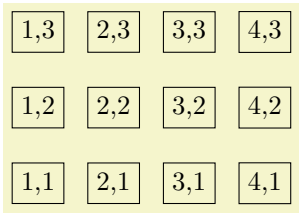
1 2 3

```
\tikz \draw (0,0) node foreach \x in {1,2,3} at (\x,0) {\x};
```

1 2 3

```
\tikz \draw (0,0) node at (1,0) {1} node at (2,0) {2} node at (3,0) {3};
```

When you provide several `foreach` statements, they work like "nested loops":



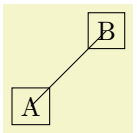

```
\tikz \node foreach \x in {1,...,4} foreach \y in {1,2,3}
            [draw] at (\x,\y) {\x,\y};
```

As the example shows, a ⟨*list*⟩ can contain ellipses (three dots) to indicate that a larger number of numbers is meant. Indeed, you can use the full power of the `\foreach` command here, including multiple parameters and options, see Section 88.

Styles for nodes. The following styles influence how nodes are rendered:

`/tikz/every node` (style, initially empty)

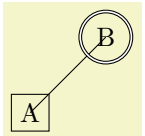
This style is installed at the beginning of every node.




```
\begin{tikzpicture}[every node/.style={draw}]
  \draw (0,0) node {A} -- (1,1) node {B};
\end{tikzpicture}
```

`/tikz/every ⟨shape⟩ node` (style, initially empty)

These styles are installed at the beginning of a node of a given ⟨*shape*⟩. For example, `every rectangle node` is used for rectangle nodes, and so on.




```
\begin{tikzpicture}
  [every rectangle node/.style={draw},
   every circle node/.style={draw,double}]
  \draw (0,0) node[rectangle] {A} -- (1,1) node[circle] {B};
\end{tikzpicture}
```

/tikz/execute at begin node=⟨*code*⟩ (no default)

This option causes ⟨*code*⟩ to be executed at the beginning of a node. Using this option multiple times will cause the code to accumulate.

/tikz/execute at end node=⟨*code*⟩ (no default)

This option installs ⟨*code*⟩ that will be executed at the end of the node. Using this option multiple times will cause the code to accumulate.

ABCD

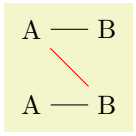
```
\begin{tikzpicture}
  [execute at begin node={A},
   execute at end node={D}]
  \node[execute at begin node={B}] {C};
\end{tikzpicture}
```

Name scopes. It turns out that the name of a node can further be influenced using two keys:

/tikz/name prefix=⟨*text*⟩ (no default, initially empty)

The value of this key is prefixed to every node inside the current scope. This includes both the naming of the node (via the `name` key or via the implicit `(`⟨*name*⟩`)` syntax) as well as any referencing of the node. Outside the scope, the nodes can (and need to) be referenced using "full name" consisting of the prefix and the node name.

The net effect of this is that you can set the name prefix at the beginning of a scope to some value and then use short and simple names for the nodes inside the scope. Later, outside the scope, you can reference the nodes via their full name:




```
\tikz {
  \begin{scope}[name prefix = top-]
    \node (A) at (0,1) {A};
    \node (B) at (1,1) {B};
    \draw (A) -- (B);
  \end{scope}
  \begin{scope}[name prefix = bottom-]
    \node (A) at (0,0) {A};
    \node (B) at (1,0) {B};
    \draw (A) -- (B);
  \end{scope}

  \draw [red] (top-A) -- (bottom-B);
}
```

As can be seen, name prefixing makes it easy to write reusable code.

/tikz/name suffix=⟨*text*⟩ (no default, initially empty)

Works as `name prefix`, only the ⟨*text*⟩ is appended to every node name in the current scope.

There is a special syntax for specifying "light-weight" nodes:

`\path` … `coordinate[`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`)` …`;`

This has the same effect as

`\node[shape=coordinate][`⟨*options*⟩`](`⟨*name*⟩`)at(`⟨*coordinate*⟩`){}`,

where the `at` part may be omitted.

Since nodes are often the only path operation on paths, there are two special commands for creating paths containing only a node:

`\node`

Inside `{tikzpicture}` this is an abbreviation for `\path node`.

`\coordinate`

Inside `{tikzpicture}` this is an abbreviation for `\path coordinate`.

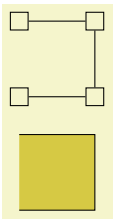
17.2.2 Predefined Shapes

PGF and Ti*k*Z define three shapes, by default:

- `rectangle`,
- `circle`, and
- `coordinate`.

By loading library packages, you can define more shapes like ellipses or diamonds; see Section 71 for the complete list of shapes.

The `coordinate` shape is handled in a special way by Ti*k*Z. When a node `x` whose shape is `coordinate` is used as a coordinate `(x)`, this has the same effect as if you had said `(x.center)`. None of the special "line shortening rules" apply in this case. This can be useful since, normally, the line shortening causes paths to be segmented and they cannot be used for filling. Here is an example that demonstrates the difference:



```
\begin{tikzpicture}[every node/.style={draw}]
  \path[yshift=1.5cm,shape=rectangle]
    (0,0) node(a1){} (1,0) node(a2){}
    (1,1) node(a3){} (0,1) node(a4){};
  \filldraw[fill=yellow!80!black] (a1) -- (a2) -- (a3) -- (a4);

  \path[shape=coordinate]
    (0,0) coordinate(b1) (1,0) coordinate(b2)
    (1,1) coordinate(b3) (0,1) coordinate(b4);
  \filldraw[fill=yellow!80!black] (b1) -- (b2) -- (b3) -- (b4);
\end{tikzpicture}
```

17.2.3 Common Options: Separations, Margins, Padding and Border Rotation

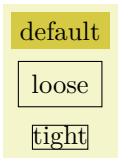
The exact behavior of shapes differs, shapes defined for more special purposes (like a, say, transistor shape) will have even more custom behaviors. However, there are some options that apply to most shapes:

`/pgf/inner sep`=⟨*dimension*⟩ (no default, initially `.3333em`)

alias `/tikz/inner sep`

An additional (invisible) separation space of ⟨*dimension*⟩ will be added inside the shape, between the text and the shape's background path. The effect is as if you had added appropriate horizontal and vertical skips at the beginning and end of the text to make it a bit "larger".

For those familiar with CSS, this is the same as *padding*.




```
\begin{tikzpicture}
  \draw (0,0)     node[inner sep=0pt,draw] {tight}
        (0cm,2em) node[inner sep=5pt,draw] {loose}
        (0cm,4em) node[fill=yellow!80!black]   {default};
\end{tikzpicture}
```

`/pgf/inner xsep`=⟨*dimension*⟩ (no default, initially `.3333em`)

alias `/tikz/inner xsep`

Specifies the inner separation in the x-direction, only.

`/pgf/inner ysep`=⟨*dimension*⟩ (no default, initially `.3333em`)

alias `/tikz/inner ysep`

Specifies the inner separation in the y-direction, only.

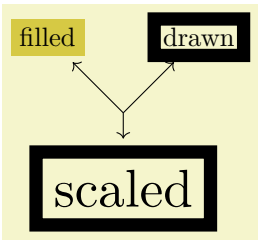
`/pgf/outer sep`=⟨*dimension or "auto"*⟩ (no default)

alias `/tikz/outer sep`

This option adds an additional (invisible) separation space of ⟨*dimension*⟩ outside the background path. The main effect of this option is that all anchors will move a little "to the outside".

For those familiar with CSS, this is same as *margin*.

The default for this option is half the line width. When the default is used and when the background path is draw, the anchors will lie exactly on the "outside border" of the path (not on the path itself).



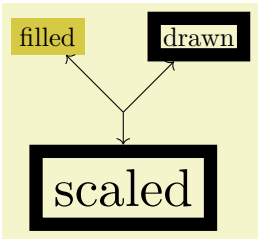

```
\begin{tikzpicture}
  \draw[line width=5pt]
    (0,0)  node[fill=yellow!80!black] (f) {filled}
    (2,0)  node[draw]                 (d) {drawn}
    (1,-2) node[draw,scale=2]         (s) {scaled};

  \draw[->] (1,-1) -- (f);
  \draw[->] (1,-1) -- (d);
  \draw[->] (1,-1) -- (s);
\end{tikzpicture}
```

As the above example demonstrates, the standard settings for the outer sep are not always "correct". First, when a shape is filled, but not drawn, the outer sep should actually be `0`. Second, when a node is scaled, for instance by a factor of 5, the outer separation also gets scaled by a factor of 5, while the line width stays at its original width; again causing problems.

In such cases, you can say `outer sep=auto` to make Ti*k*Z *try* to compensate for the effects described above. This is done by, firstly, setting the outer sep to `0` when no drawing is done and, secondly, setting the outer separations to half the line width (as before) times two adjustment factors, one for the horizontal separations and one for the vertical separations (see Section 108.2.6 for details on these factors). Note, however, that these factors can compensate only for transformations that are either scalings plus rotations or scalings with different magnitudes in the horizontal and the vertical direction. If you apply slanting, the factors will only approximate the correct values.

In general, it is a good idea to say `outer sep=auto` at some early stage. It is not the default mainly for compatibility with earlier versions.




```
\begin{tikzpicture}[outer sep=auto]
  \draw[line width=5pt]
    (0,0)  node[fill=yellow!80!black] (f) {filled}
    (2,0)  node[draw]                 (d) {drawn}
    (1,-2) node[draw,scale=2]         (s) {scaled};

  \draw[->] (1,-1) -- (f);
  \draw[->] (1,-1) -- (d);
  \draw[->] (1,-1) -- (s);
\end{tikzpicture}
```

`/pgf/outer xsep=`⟨*dimension*⟩ (no default, initially `.5\pgflinewidth`)

alias `/tikz/outer xsep`

Specifies the outer separation in the x-direction, only. This value will be overwritten when `outer sep` is set, either to the value given there or a computed value in case of `auto`.

`/pgf/outer ysep=`⟨*dimension*⟩ (no default, initially `.5\pgflinewidth`)

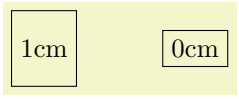
alias `/tikz/outer ysep`

Specifies the outer separation in the y-direction, only.

`/pgf/minimum height=`⟨*dimension*⟩ (no default, initially `1pt`)

alias `/tikz/minimum height`

This option ensures that the height of the shape (including the inner, but ignoring the outer separation) will be at least ⟨*dimension*⟩. Thus, if the text plus the inner separation is not at least as large as ⟨*dimension*⟩, the shape will be enlarged appropriately. However, if the text is already larger than ⟨*dimension*⟩, the shape will not be shrunk.

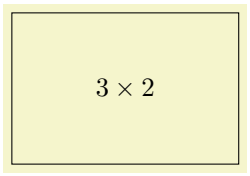



```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=1cm,draw] {1cm}
        (2,0) node[minimum height=0cm,draw] {0cm};
\end{tikzpicture}
```

`/pgf/minimum width=`⟨*dimension*⟩ (no default, initially `1pt`)

alias `/tikz/minimum width`

Same as `minimum height`, only for the width.

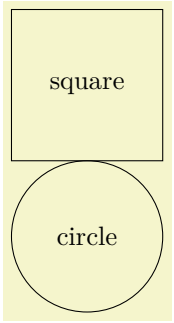



```
\begin{tikzpicture}
  \draw (0,0) node[minimum height=2cm,minimum width=3cm,draw] {$3 \times 2$};
\end{tikzpicture}
```

/pgf/minimum size=⟨*dimension*⟩ (no default)

alias /tikz/minimum size

Sets both the minimum height and width at the same time.

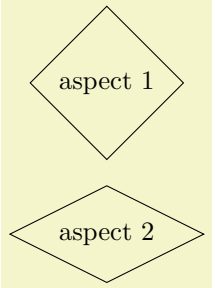



```
\begin{tikzpicture}
  \draw (0,0)  node[minimum size=2cm,draw] {square};
  \draw (0,-2) node[minimum size=2cm,draw,circle] {circle};
\end{tikzpicture}
```

/pgf/shape aspect=⟨*aspect ratio*⟩ (no default)

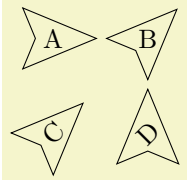
alias /tikz/shape aspect

Sets a desired aspect ratio for the shape. For the diamond shape, this option sets the ratio between width and height of the shape.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \draw (0,0)  node[shape aspect=1,diamond,draw] {aspect 1};
  \draw (0,-2) node[shape aspect=2,diamond,draw] {aspect 2};
\end{tikzpicture}
```

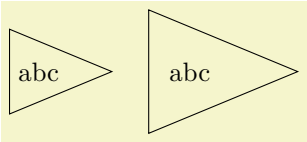
Some shapes (but not all), support a special kind of rotation. This rotation affects only the border of a shape and is independent of the node contents, but *in addition* to any other transformations.




```
\usetikzlibrary {shapes.geometric}
\tikzset{every node/.style={dart, shape border uses incircle,
  inner sep=1pt, draw}}
\tikz \node foreach \a/\b/\c in {A/0/0, B/45/0, C/0/45, D/45/45}
            [shape border rotate=\b, rotate=\c] at (\b/36,-\c/36) {\a};
```

There are two types of rotation: restricted and unrestricted. Which type of rotation is applied is determined by on how the shape border is constructed. If the shape border is constructed using an incircle, that is, a circle that tightly fits the node contents (including the inner sep), then the rotation can be unrestricted. If, however, the border is constructed using the natural dimensions of the node contents, the rotation is restricted to integer multiples of 90 degrees.

Why should there be two kinds of rotation and border construction? Borders constructed using the natural dimensions of the node contents provide a much tighter fit to the node contents, but to maintain this tight fit, the border rotation must be restricted to integer multiples of 90 degrees. By using an incircle, unrestricted rotation is possible, but the border will not make a very tight fit to the node contents.




```
\usetikzlibrary {shapes.geometric}
\tikzset{every node/.style={isosceles triangle, draw}}
\begin{tikzpicture}
  \node {abc};
  \node [shape border uses incircle] at (2,0) {abc};
\end{tikzpicture}
```

There are PGF keys that determine how a shape border is constructed, and to specify its rotation. It should be noted that not all shapes support these keys, so reference should be made to the documentation for individual shapes.

`/pgf/shape border uses incircle=`⟨*boolean*⟩ (default `true`)

alias `/tikz/shape border uses incircle`

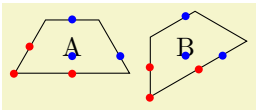
Determines if the border of a shape is constructed using the incircle. If no value is given ⟨*boolean*⟩ will take the default value `true`.

`/pgf/shape border rotate=`⟨*angle*⟩ (no default, initially `0`)

alias `/tikz/shape border rotate`

Rotates the border of a shape independently of the node contents, but in addition to any other transformations. If the shape border is not constructed using the incircle, the rotation will be rounded to the nearest integer multiple of 90 degrees when the shape is drawn.

Note that if the border of the shape is rotated, the compass point anchors, and 'text box' anchors (including `mid east`, `base west`, and so on), *do not rotate*, but the other anchors do:




```
\usetikzlibrary {shapes.geometric}
\tikzset{every node/.style={shape=trapezium, draw, shape border uses incircle}}
\begin{tikzpicture}
  \node at (0,0)   (A) {A};
  \node [shape border rotate=30] at (1.5,0) (B) {B};
  \foreach \s/\t in
    {left side/base east, bottom side/north, bottom left corner/base}{
       \fill[red]  (A.\s) circle[radius=1.5pt] (B.\s) circle[radius=1.5pt];
       \fill[blue] (A.\t) circle[radius=1.5pt] (B.\t) circle[radius=1.5pt];
  }
\end{tikzpicture}
```

Finally, a somewhat unfortunate side-effect of rotating shape borders is that the supporting shapes do not distinguish between `outer xsep` and `outer ysep`, and typically, the larger of the two values will be used.

17.3 Multi-Part Nodes

Most nodes just have a single simple text label. However, nodes of a more complicated shape might be made up from several *node parts*. For example, in automata theory a so-called Moore state has a state name, drawn in the upper part of the state circle, and an output text, drawn in the lower part of the state circle. These two parts are quite independent. Similarly, a UML class shape would have a name part, a method part, and an attributes part. Different molecule shapes might use parts for the different atoms to be drawn at the different positions, and so on.

Both PGF and Ti*k*Z support such multipart nodes. On the lower level, PGF provides a system for specifying that a shape consists of several parts. On the Ti*k*Z level, you specify the different node parts by using the following command:

`\nodepart[`⟨*options*⟩`]{`⟨*part name*⟩`}`

This command can only be used inside the ⟨*text*⟩ argument of a `node` path operation. It works a little bit like a `\part` command in LaTeX. It will stop the typesetting of whatever node part was typeset until now and then start putting all following text into the node part named ⟨*part name*⟩ – until another `\nodepart` is encountered or until the node ⟨*text*⟩ ends. The ⟨*options*⟩ will be local to this part.




```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
  \node [circle split,draw,double,fill=red!20]
  {
    % No \nodepart has been used, yet. So, the following is put in the
    % ``text'' node part by default.
    $q_1$
    \nodepart{lower} % Ok, end ``text'' part, start ``output'' part
    $00$
  }; % output part ended.
\end{tikzpicture}
```

You will have to lookup which parts are defined by a shape.

The following styles influences node parts:

`/tikz/every ⟨part name⟩ node part` (style, initially empty)

This style is installed at the beginning of every node part named ⟨*part name*⟩.




```
\usetikzlibrary {shapes.multipart}
\tikz [every lower node part/.style={red}]
  \node [circle split,draw] {$q_1$ \nodepart{lower} $00$};
```

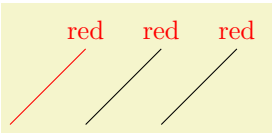
17.4 The Node Text

17.4.1 Text Parameters: Color and Opacity

The simplest option for the text in nodes is its color. Normally, this color is just the last color installed using `color=`, possibly inherited from another scope. However, it is possible to specifically set the color used for text using the following option:

`/tikz/text=`⟨*color*⟩ (no default)

Sets the color to be used for text labels. A `color=` option will immediately override this option.




```
\begin{tikzpicture}
  \draw[red]       (0,0) -- +(1,1) node[above]     {red};
  \draw[text=red]  (1,0) -- +(1,1) node[above]     {red};
  \draw            (2,0) -- +(1,1) node[above,red] {red};
\end{tikzpicture}
```

Just like the color itself, you may also wish to set the opacity of the text only. For this, use the `text opacity` option, which is detailed in Section 23.

17.4.2 Text Parameters: Font

Next, you may wish to adjust the font used for the text. Naturally, you can just use a font command like `\small` or `\rm` at the beginning of a node. However, the following two options make it easier to set the font used in nodes on a general basis. Let us start with:

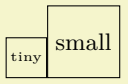
`/tikz/node font=`⟨*font commands*⟩ (no default)

This option sets the font used for all text used in a node.




```
\begin{tikzpicture}
  \draw[node font=\itshape] (1,0) -- +(1,1) node[above] {italic};
\end{tikzpicture}
```

Since the ⟨*font commands*⟩ are executed at a very early stage in the construction of the node, the font selected using this command will also dictate the values of dimensions defined in terms of `em` or `ex`. For instance, when the `minimum height` of a node is `3em`, the actual height will be (at least) three times the line distance selected by the ⟨*font commands*⟩:




```
\tikz \node [node font=\tiny,  minimum height=3em, draw] {tiny};
\tikz \node [node font=\small, minimum height=3em, draw] {small};
```

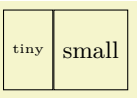
The other font command is:

/tikz/font=⟨*font commands*⟩ (no default)

Sets the font used for the text inside nodes. However, this font will *not* (yet) be installed when any of the dimensions of the node are being computed, so dimensions like `1em` will be with respect to the font used outside the node (usually the font that was in force when the picture started).

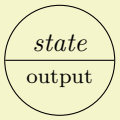
italic

```
\begin{tikzpicture}
  \node [font=\itshape] {italic};
\end{tikzpicture}
```




```
\tikz \node [font=\tiny,  minimum height=3em, draw] {tiny};
\tikz \node [font=\small, minimum height=3em, draw] {small};
```

A useful example of how the `font` option can be used is the following:




```
\usetikzlibrary {shapes.multipart}
\tikz [every text node part/.style={font=\itshape},
       every lower node part/.style={font=\footnotesize}]
  \node [circle split,draw] {state \nodepart{lower} output};
```

As can be seen, the font can be changed for each node part. This does *not* work with the `node font` command since, as the name suggests, this command can only be used to select the "overall" font for the node and this is done very early.

17.4.3 Text Parameters: Alignment and Width for Multi-Line Text

Normally, when a node is typeset, all the text you give in the braces is put in one long line (in an `\hbox`, to be precise) and the node will become as wide as necessary.

From time to time you may wish to create nodes that contain multiple lines of text. There are three different ways of achieving this:

1. Inside the node, you can put some standard environment that produces multi-line, aligned text. For instance, you can use a `{tabular}` inside a node:

upper left	upper right
lower left	lower right

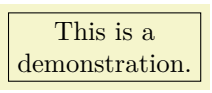
```
\tikz \node [draw] {
  \begin{tabular}{cc}
    upper left & upper right\\
    lower left & lower right
  \end{tabular}
};
```

This approach offers the most flexibility in the sense that it allows you to use all of the alignment commands offered by your format of choice.

2. You use `\\` inside your node to mark the end of lines and then request Ti*k*Z to arrange these lines in some manner. This will only be done, however, if the `align` option has been given.

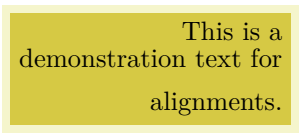
This is a
demonstration.

```
\tikz[align=left] \node[draw] {This is a\\demonstration.};
```




```
\tikz[align=center] \node[draw] {This is a\\demonstration.};
```

The \\ command takes an optional extra space as an argument in square brackets.




```
\tikz \node[fill=yellow!80!black,align=right]
  {This is a\\[-2pt] demonstration text for\\[1ex] alignments.};
```

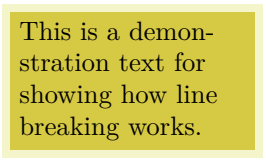
3. You can request that Ti*k*Z does an automatic line-breaking for you inside the node by specifying a fixed `text width` for the node. In this case, you can still use \\ to enforce a line-break. Note that when you specify a text width, the node will have this width, independently of whether the text actually "reaches the end" of the node.

Let us now first have a look at the `text width` command.

`/tikz/text width`=⟨*dimension*⟩ (no default)

This option will put the text of a node in a box of the given width (something akin to a `{minipage}` of this width, only portable across formats). If the node text is not as wide as ⟨*dimension*⟩, it will nevertheless be put in a box of this width. If it is larger, line breaking will be done.

By default, when this option is given, a ragged right border will be used (`align=left`). This is sensible since, typically, these boxes are narrow and justifying the text looks ugly. You can, however, change the alignment using `align` or directly using commands line `\centering`.




```
\tikz \draw (0,0) node[fill=yellow!80!black,text width=3cm]
  {This is a demonstration text for showing how line breaking works.};
```

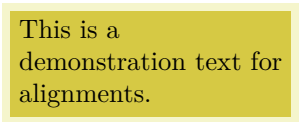
Setting ⟨*dimension*⟩ to an empty string causes the automatic line breaking to be disabled.

`/tikz/align`=⟨*alignment option*⟩ (no default)

This key is used to set up an alignment for multi-line text inside a node. If `text width` is set to some width (let us call this *alignment with line breaking*), the `align` key will setup the `\leftskip` and the `\rightskip` in such a way that the text is broken and aligned according to ⟨*alignment option*⟩. If `text width` is not set (that is, set to the empty string; let us call this *alignment without line breaking*), then a different mechanism is used internally, namely the key `node halign header`, is set to an appropriate value. While this key, which is documented below, is not to be used by beginners, the net effect is simple: When `text width` is not set, you can use \\ to break lines and align them according to ⟨*alignment option*⟩ and the resulting node's width will be minimal to encompass the resulting lines.

In detail, you can set ⟨*alignment option*⟩ to one of the following values:

`align=left` For alignment without line breaking, the different lines are simply aligned such that their left borders are below one another.




```
\tikz \node[fill=yellow!80!black,align=left]
  {This is a\\ demonstration text for\\ alignments.};
```

For alignment with line breaking, the same will happen; only the lines will now, additionally, be broken automatically:

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=left]
  {This is a demonstration text for showing how line breaking works.};
```

align=flush left For alignment without line breaking this option has exactly the same effect as **left**. However, for alignment with line breaking, there is a difference: While **left** uses the original plain TEX definition of a ragged right border, in which TEX will try to balance the right border as well as possible, **flush left** causes the right border to be ragged in the LATEX-style, in which no balancing occurs. This looks ugly, but it may be useful for very narrow boxes and when you wish to avoid hyphenations.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=flush left]
  {This is a demonstration text for showing how line breaking works.};
```

align=right Works like **left**, only for right alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=yellow!80!black,align=right]
  {This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=right]
  {This is a demonstration text for showing how line breaking works.};
```

align=flush right Works like **flush left**, only for right alignment.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=flush right]
  {This is a demonstration text for showing how line breaking works.};
```

align=center Works like **left** or **right**, only for centered alignment.

This is a demonstration text for alignments.

```
\tikz \node[fill=yellow!80!black,align=center]
  {This is a\\ demonstration text for\\ alignments.};
```

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=center]
  {This is a demonstration text for showing how line breaking works.};
```

There is one annoying problem with the `center` alignment (but not with `flush center` and the other options): If you specify a large line width and the node text fits on a single line and is, in fact, much shorter than the specified `text width`, an underfull horizontal box will result. Unfortunately, this cannot be avoided, due to the way TEX works (more precisely, I have thought long and hard about this and have not been able to figure out a sensible way to avoid this). For this reason, Ti*k*Z switches off horizontal badness warnings inside boxes with `align=center`. Since this will also suppress some "wanted" warnings, there is also an option for switching the warnings on once more:

`/tikz/badness warnings for centered text=`⟨*true or false*⟩ (no default, initially `false`)

If set to true, normal badness warnings will be issued for centered boxes. Note that you may get annoying warnings for perfectly normal boxes, namely whenever the box is very large and the contents is not long enough to fill the box sufficiently.

`align=flush center` Works like `flush left` or `flush right`, only for center alignment. Because of all the trouble that results from the `center` option in conjunction with narrow lines, I suggest picking this option rather than `center` *unless* you have longer text, in which case `center` will give the typographically better results.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=flush center]
  {This is a demonstration text for showing how line breaking works.};
```

`align=justify` For alignment without line breaking, this has the same effect as `left`. For alignment with line breaking, this causes the text to be "justified". Use this only with rather broad nodes.

This is a demonstration text for showing how line breaking works.

```
\tikz \node[fill=yellow!80!black,text width=3cm,align=justify]
  {This is a demonstration text for showing how line breaking works.};
```

In the above example, TEX complains (rightfully) about three very badly typeset lines. (For this manual I asked TEX to stop complaining by using `\hbadness=10000`, but this is a foul deed, indeed.)

`align=none` Disables all alignments and `\\` will not be redefined.

`/tikz/node halign header=`⟨*macro storing a header*⟩ (no default, initially empty)

This is the key that is used by `align` internally for alignment without line breaking. Read the following only if you are familiar with the `\halign` command.

This key only has an effect if `text width` is empty, otherwise it is ignored. Furthermore, if ⟨*macro storing a header*⟩ is empty, then this key also has no effect. So, suppose `text width` is empty, but ⟨*header*⟩ is not. In this case the following happens:

When the node text is parsed, the command `\\` is redefined internally. This redefinition is done in such a way that the text from the start of the node to the first occurrence of `\\` is put in an `\hbox`. Then the text following `\\` up to the next `\\` is put in another `\hbox`. This goes on until the text between the last `\\` and the closing `}` is also put in an `\hbox`.

The ⟨*macro storing a header*⟩ should be a macro that contains some text suitable for use as a header for the `\halign` command. For instance, you might define

```
\def\myheader{\hfil\hfil##\hfil\cr}
\tikz [node halign header=\myheader] ...
```

You cannot just say `node halign header=\hfil\hfil#\hfil\cr` because this confuses TEX inside matrices, so this detour via a macro is needed.

Next, conceptually, all these boxes are recursively put inside an `\halign` command. Assuming that ⟨*first*⟩ is the first of the above boxes, the command `\halign{`⟨*header*⟩ `\box`⟨*first*⟩ `\cr}` is used to create a new box, which we will call the ⟨*previous box*⟩. Then, the following box is created, where ⟨*second*⟩ is the second input box: `\halign{`⟨*header*⟩ `\box`⟨*previous box*⟩ `\cr \box`⟨*second*⟩`\cr}`. Let us call the resulting box the ⟨*previous box*⟩ once more. Then the next box that is created is `\halign{`⟨*header*⟩ `\box`⟨*previous box*⟩ `\cr \box`⟨*third*⟩`\cr}`.

All of this means that if ⟨*header*⟩ is an `\halign` header like `\hfil#\hfil\cr`, then all boxes will be centered relative to one another. Similarly, a ⟨*header*⟩ of `\hfil#\cr` causes the text to be flushed right.

Note that this mechanism is not flexible enough to all multiple columns inside ⟨*header*⟩. You will have to use a `tabular` or a `matrix` in such cases.

One further note: Since the text of each line is placed in a box, settings will be local to each "line". This is very similar to the way a cell in a `tabular` or a `matrix` behaves.

17.4.4 Text Parameters: Height and Depth of Text

In addition to changing the width of nodes, you can also change the height of nodes. This can be done in two ways: First, you can use the option `minimum height`, which ensures that the height of the whole node is at least the given height (this option is described in more detail later). Second, you can use the option `text height`, which sets the height of the text itself, more precisely, of the TEX text box of the text. Note that the `text height` typically is not the height of the shape's box: In addition to the `text height`, an internal `inner sep` is added as extra space and the text depth is also taken into account.

I recommend using `minimum size` instead of `text height` except for special situations.

`/tikz/text height=`⟨*dimension*⟩ (no default)

Sets the height of the text boxes in shapes. Thus, when you write something like `node {text}`, the `text` is first typeset, resulting in some box of a certain height. This height is then replaced by the height `text height`. The resulting box is then used to determine the size of the shape, which will typically be larger. When you write `text height=` without specifying anything, the "natural" size of the text box remains unchanged.




```
\tikz \node[draw]                    {y};
\tikz \node[draw,text height=10pt] {y};
```

`/tikz/text depth=`⟨*dimension*⟩ (no default)

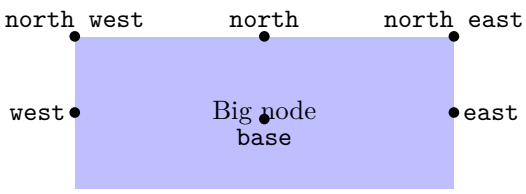
This option works like `text height`, only for the depth of the text box. This option is mostly useful when you need to ensure a uniform depth of text boxes that need to be aligned.

17.5 Positioning Nodes

When you place a node at some coordinate, the node is centered on this coordinate by default. This is often undesirable and it would be better to have the node to the right or above the actual coordinate.

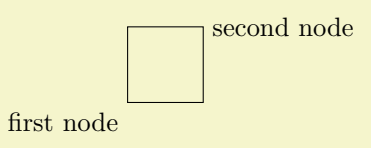
17.5.1 Positioning Nodes Using Anchors

PGF uses a so-called anchoring mechanism to give you a very fine control over the placement. The idea is simple: Imagine a node of rectangular shape of a certain size. PGF defines numerous anchor positions in the shape. For example the upper right corner is called, well, not "upper right anchor", but the `north east` anchor of the shape. The center of the shape has an anchor called `center` on top of it, and so on. Here are some examples (a complete list for predefined shapes is given in Section 71.2).



Now, when you place a node at a certain coordinate, you can ask TikZ to place the node shifted around in such a way that a certain anchor is at the coordinate. In the following example, we ask TikZ to shift the

first node such that its `north east` anchor is at coordinate `(0,0)` and that the `west` anchor of the second node is at coordinate `(1,1)`.




```
\tikz \draw          (0,0) node[anchor=north east] {first node}
           rectangle (1,1) node[anchor=west] {second node};
```

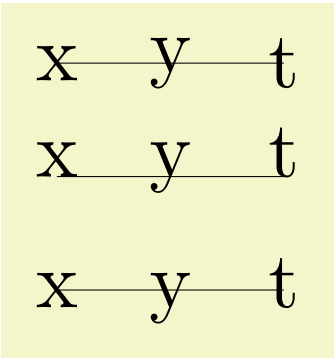
Since the default anchor is `center`, the default behavior is to shift the node in such a way that it is centered on the current position.

`/tikz/anchor=`⟨*anchor name*⟩ (no default)

Causes the node to be shifted such that its anchor ⟨*anchor name*⟩ lies on the current coordinate.

The only anchor that is present in all shapes is `center`. However, most shapes will at least define anchors in all "compass directions". Furthermore, the standard shapes also define a `base` anchor, as well as `base west` and `base east`, for placing things on the baseline of the text.

The standard shapes also define a `mid` anchor (and `mid west` and `mid east`). This anchor is half the height of the character "x" above the base line. This anchor is useful for vertically centering multiple nodes that have different heights and depth. Here is an example:




```
\begin{tikzpicture}[scale=3,transform shape]
  % First, center alignment -> wobbles
  \draw[anchor=center] (0,1)  node{x} -- (0.5,1)  node{y} -- (1,1)  node{t};
  % Second, base alignment -> no wobble, but too high
  \draw[anchor=base]   (0,.5) node{x} -- (0.5,.5) node{y} -- (1,.5) node{t};
  % Third, mid alignment
  \draw[anchor=mid]    (0,0)  node{x} -- (0.5,0)  node{y} -- (1,0)  node{t};
\end{tikzpicture}
```

17.5.2 Basic Placement Options

Unfortunately, while perfectly logical, it is often rather counter-intuitive that in order to place a node *above* a given point, you need to specify the `south` anchor. For this reason, there are some useful options that allow you to select the standard anchors more intuitively:

`/tikz/above=`⟨*offset*⟩ (default `0pt`)

Does the same as `anchor=south`. If the ⟨*offset*⟩ is specified, the node is additionally shifted upwards by the given ⟨*offset*⟩.

above

```
\tikz \fill (0,0) circle [radius=2pt] node[above] {above};
```

above

```
\tikz \fill (0,0) circle [radius=2pt] node[above=2pt] {above};
```

/tikz/below=⟨*offset*⟩ (default 0pt)

Similar to above.

/tikz/left=⟨*offset*⟩ (default 0pt)

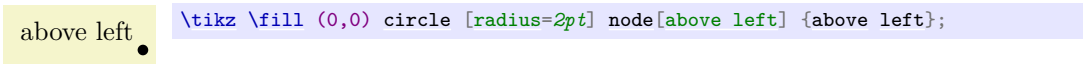
Similar to above.

/tikz/right=⟨*offset*⟩ (default 0pt)

Similar to above.

/tikz/above left (no value)

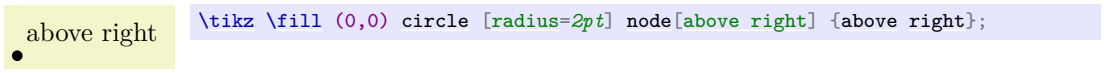
Does the same as anchor=south east. Note that giving both above and left options does not have the same effect as above left, rather only the last left "wins". Actually, this option also takes an ⟨*offset*⟩ parameter, but using this parameter without using the positioning library is deprecated. (The positioning library changes the meaning of this parameter to something more sensible.)




```
\tikz \fill (0,0) circle [radius=2pt] node[above left] {above left};
```

/tikz/above right (no value)

Similar to above left.




```
\tikz \fill (0,0) circle [radius=2pt] node[above right] {above right};
```

/tikz/below left (no value)

Similar to above left.

/tikz/below right (no value)

Similar to above left.

/tikz/centered (no value)

A shorthand for anchor=center.

17.5.3 Advanced Placement Options

While the standard placement options suffice for simple cases, the positioning library offers more convenient placement options.

TikZ Library positioning

```
\usetikzlibrary{positioning} % LaTeX and plain TeX
\usetikzlibrary[positioning] % ConTeXt
```

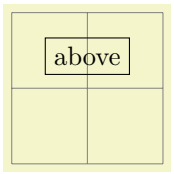
The library defines additional options for placing nodes conveniently. It also redefines the standard options like above so that they give you better control of node placement.

When this library is loaded, the options like above or above left behave differently.

/tikz/above=⟨*specification*⟩ (default 0pt)

With the positioning library loaded, the above option does not take a simple ⟨*dimension*⟩ as its parameter. Rather, it can (also) take a more elaborate ⟨*specification*⟩ as parameter. This ⟨*specification*⟩ has the following general form: It starts with an optional ⟨*shifting part*⟩ and is followed by an optional ⟨*of-part*⟩. Let us start with the ⟨*shifting part*⟩, which can have three forms:

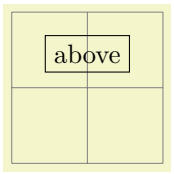
1. It can simply be a ⟨*dimension*⟩ (or a mathematical expression that evaluates to a dimension) like 2cm or 3cm/2+4cm. In this case, the following happens: the node's anchor is set to south and the node is vertically shifted upwards by ⟨*dimension*⟩.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=2pt+3pt,draw] {above};
\end{tikzpicture}
```

This use of the `above` option is the same as if the `positioning` library were not loaded.

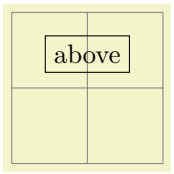
2. It can be a ⟨*number*⟩ (that is, any mathematical expression that does not include a unit like `pt` or `cm`). Examples are `2` or `3+sin(60)`. In this case, the anchor is also set to `south` and the node is vertically shifted by the vertical component of the coordinate `(0,`⟨*number*⟩`)`.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=.2,draw] {above};
  % south border of the node is now 2mm above (1,1)
\end{tikzpicture}
```

3. It can be of the form ⟨*number or dimension 1*⟩ `and` ⟨*number or dimension 2*⟩. This specification does not make particular sense for the `above` option, it is much more useful for options like `above left`. The reason it is allowed for the `above` option is that it is sometimes automatically used, as explained later.

   The effect of this option is the following. First, the point `(`⟨*number or dimension 2*⟩`,`⟨*number or dimension 1*⟩`)` is computed (note the inverted order), using the normal rules for evaluating such a coordinate, yielding some position. Then, the node is shifted by the vertical component of this point. The anchor is set to `south`.

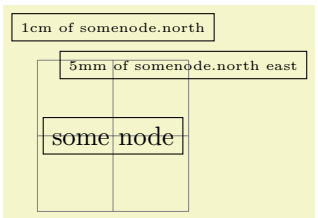



```
\usetikzlibrary {positioning}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \node at (1,1) [above=.2 and 3mm,draw] {above};
  % south border of the node is also 2mm above (1,1)
\end{tikzpicture}
```

The ⟨*shifting part*⟩ can optionally be followed by a ⟨*of-part*⟩, which has one of the following forms:

1. The ⟨*of-part*⟩ can be `of` ⟨*coordinate*⟩, where ⟨*coordinate*⟩ is *not* in parentheses and it is *not* just a node name. An example would be `of somenode.north` or `of {2,3}`. In this case, the following happens: First, the node's `at` parameter is set to the ⟨*coordinate*⟩. Second, the node is shifted according to the ⟨*shift-part*⟩. Third, the anchor is set to `south`.

   Here is a basic example:




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,2);
  \node (somenode) at (1,1) {some node};

  \node [above=5mm of somenode.north east] {\tiny 5mm of somenode.north east};
  \node [above=1cm of somenode.north]      {\tiny 1cm of somenode.north};
\end{tikzpicture}
```

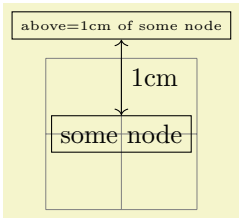
   As can be seen the `above=5mm of somenode.north east` option does, indeed, place the node 5mm above the north east anchor of `somenode`. The same effect could have been achieved writing `above=5mm` followed by `at=(somenode.north east)`.

   If the ⟨*shifting-part*⟩ is missing, the shift is not zero, but rather the value of the `node distance` key is used, see below.

2. The ⟨*of-part*⟩ can be `of` ⟨*node name*⟩. An example would be `of somenode`. In this case, the following usually happens:

- The anchor is set to `south`.
- The node is shifted according to the ⟨*shifting part*⟩ or, if it is missing, according to the value of `node distance`.
- The node's `at` parameter is set to ⟨*node name*⟩`.north`.

The net effect of all this is that the new node will be placed in such a way that the distance between its south border and ⟨*node name*⟩'s north border is exactly the given distance.




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,2);
  \node (some node) at (1,1) {some node};

  \node (other node) [above=1cm of some node] {\tiny above=1cm of some node};

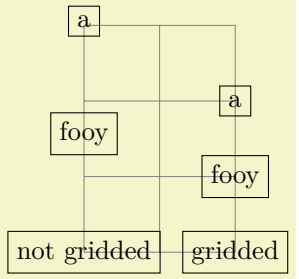
  \draw [<->] (some node.north) -- (other node.south)
                                node [midway,right,draw=none] {1cm};
\end{tikzpicture}
```

It is possible to change the behavior of this ⟨*specification*⟩ rather drastically, using the following key:

`/tikz/on grid`=⟨*boolean*⟩ (no default, initially `false`)

When this key is set to `true`, an ⟨*of-part*⟩ of the current form behaves differently: The anchors set for the current node as well as the anchor used for the other ⟨*node name*⟩ are set to `center`.

This has the following effect: When you say `above=1cm of somenode` with `on grid` set to true, the new node will be placed in such a way that its center is 1cm above the center of `somenode`. Repeatedly placing nodes in this way will result in nodes that are centered on "grid coordinate", hence the name of the option.



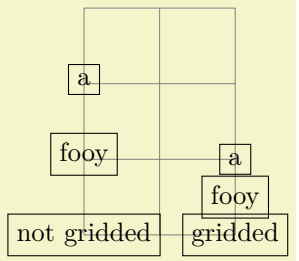

```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw]
  \draw[help lines] (0,0) grid (2,3);

  % Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=1cm of a1] {fooy};
  \node (c1) [above=1cm of b1] {a};

  % gridded
  \node (a2) at (2,0) {gridded};
  \node (b2) [on grid,above=1cm of a2] {fooy};
  \node (c2) [on grid,above=1cm of b2] {a};
\end{tikzpicture}
```

`/tikz/node distance`=⟨*shifting part*⟩ (no default, initially `1cm and 1cm`)

The value of this key is used as ⟨*shifting part*⟩ is used if and only if a ⟨*of-part*⟩ is present, but no ⟨*shifting part*⟩.




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style=draw,node distance=5mm]
  \draw[help lines] (0,0) grid (2,3);

  % Not gridded
  \node (a1) at (0,0) {not gridded};
  \node (b1) [above=of a1] {fooy};
  \node (c1) [above=of b1] {a};

  % gridded
  \begin{scope}[on grid]
    \node (a2) at (2,0) {gridded};
    \node (b2) [above=of a2] {fooy};
    \node (c2) [above=of b2] {a};
  \end{scope}
\end{tikzpicture}
```

`/tikz/below`=⟨*specification*⟩ (no default)

This key is redefined in the same manner as `above`.

`/tikz/left=`⟨*specification*⟩ (no default)

This key is redefined in the same manner as `above`, only all vertical shifts are replaced by horizontal shifts.

`/tikz/right=`⟨*specification*⟩ (no default)

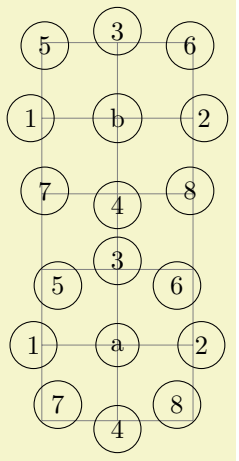
This key is redefined in the same manner as `left`.

`/tikz/above left=`⟨*specification*⟩ (no default)

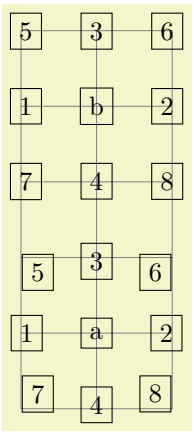
This key is also redefined in a manner similar to the above, but behavior of the ⟨*shifting part*⟩ is more complicated:

1. When the ⟨*shifting part*⟩ is of the form ⟨*number or dimension*⟩ `and` ⟨*number or dimension*⟩, it has (essentially) the effect of shifting the node vertically upwards by the first ⟨*number or dimension*⟩ and to the left by the second. To be more precise, the coordinate `(`⟨*second number or dimension*⟩`,`⟨*first number or dimension*⟩`)` is computed and then the node is shifted vertically by the y-part of the resulting coordinate and horizontally be the negated x-part of the result. (This is exactly what you expect, except possibly when you have used the `x` and `y` options to modify the `xy`-coordinate system so that the unit vectors no longer point in the expected directions.)
2. When the ⟨*shifting part*⟩ is of the form ⟨*number or dimension*⟩, the node is shifted by this ⟨*number or dimension*⟩ in the direction of 135°. This means that there is a difference between a ⟨*shifting part*⟩ of `1cm` and of `1cm and 1cm`: In the second case, the node is shifted by 1cm upward and 1cm to the left; in the first case it is shifted by $\frac{1}{2}\sqrt{2}$cm upward and by the same amount to the left. A more mathematical way of phrasing this is the following: A plain ⟨*dimension*⟩ is measured in the l_2-norm, while a ⟨*dimension*⟩ `and` ⟨*dimension*⟩ is measured in the l_1-norm.

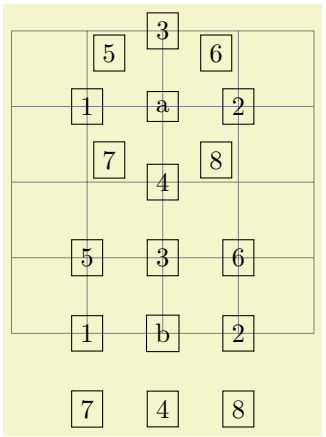
The following example should help to illustrate the difference:




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,circle}]
  \draw[help lines] (0,0) grid (2,5);
  \begin{scope}[node distance=5mm and 5mm]
    \node (b) at (1,4) {b};
    \node [left=of b] {1};        \node [right=of b] {2};
    \node [above=of b] {3};       \node [below=of b] {4};
    \node [above left=of b] {5}; \node [above right=of b] {6};
    \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
  \begin{scope}[node distance=5mm]
    \node (a) at (1,1) {a};
    \node [left=of a] {1};        \node [right=of a] {2};
    \node [above=of a] {3};       \node [below=of a] {4};
    \node [above left=of a] {5}; \node [above right=of a] {6};
    \node [below left=of a] {7}; \node [below right=of a] {8};
  \end{scope}
\end{tikzpicture}
```




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,rectangle}]
  \draw[help lines] (0,0) grid (2,5);
  \begin{scope}[node distance=5mm and 5mm]
    \node (b) at (1,4) {b};
    \node [left=of b] {1};        \node [right=of b] {2};
    \node [above=of b] {3};       \node [below=of b] {4};
    \node [above left=of b] {5}; \node [above right=of b] {6};
    \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
  \begin{scope}[node distance=5mm]
    \node (a) at (1,1) {a};
    \node [left=of a] {1};        \node [right=of a] {2};
    \node [above=of a] {3};       \node [below=of a] {4};
    \node [above left=of a] {5}; \node [above right=of a] {6};
    \node [below left=of a] {7}; \node [below right=of a] {8};
  \end{scope}
\end{tikzpicture}
```




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[every node/.style={draw,rectangle},on grid]
  \draw[help lines] (0,0) grid (4,4);
  \begin{scope}[node distance=1]
    \node (a) at (2,3) {a};
    \node [left=of a] {1};        \node [right=of a] {2};
    \node [above=of a] {3};       \node [below=of a] {4};
    \node [above left=of a] {5}; \node [above right=of a] {6};
    \node [below left=of a] {7}; \node [below right=of a] {8};
  \end{scope}
  \begin{scope}[node distance=1 and 1]
    \node (b) at (2,0) {b};
    \node [left=of b] {1};        \node [right=of b] {2};
    \node [above=of b] {3};       \node [below=of b] {4};
    \node [above left=of b] {5}; \node [above right=of b] {6};
    \node [below left=of b] {7}; \node [below right=of b] {8};
  \end{scope}
\end{tikzpicture}
```

/tikz/below left=⟨*specification*⟩ (no default)

Works similar to `above left`.

/tikz/above right=⟨*specification*⟩ (no default)

Works similar to `above left`.

/tikz/below right=⟨*specification*⟩ (no default)

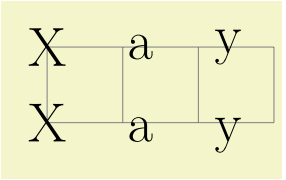
Works similar to `above left`.

The `positioning` package also introduces the following new placement keys:

/tikz/base left=⟨*specification*⟩ (no default)

This key works like the `left` key, only instead of the `east` anchor, the `base east` anchor is used and, when the second form of an ⟨*of-part*⟩ is used, the corresponding `base west` anchor.

This key is useful for chaining together nodes so that their base lines are aligned.




```
\usetikzlibrary {positioning}
\begin{tikzpicture}[node distance=1ex]
  \draw[help lines] (0,0) grid (3,1);
  \huge
  \node (X) at (0,1)      {X};
  \node (a) [right=of X] {a};
  \node (y) [right=of a] {y};

  \node (X) at (0,0)          {X};
  \node (a) [base right=of X] {a};
  \node (y) [base right=of a] {y};
\end{tikzpicture}
```

/tikz/base right=⟨*specification*⟩ (no default)

Works like `base left`.

/tikz/mid left=⟨*specification*⟩ (no default)

Works like `base left`, but with `mid east` and `mid west` anchors instead of `base east` and `base west`.

/tikz/mid right=⟨*specification*⟩ (no default)

Works like `mid left`.

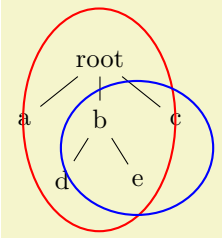
17.5.4 Advanced Arrangements of Nodes

The simple `above` and `right` options may not always suffice for arranging a large number of nodes. For such situations Ti*k*Z offers libraries that make positioning easier: The `matrix` library and the `graphdrawing` library. These libraries for positioning nodes are described in two separate Sections 20 and 27.

17.6 Fitting Nodes to a Set of Coordinates

It is sometimes desirable that the size and position of a node is not given using anchors and size parameters, rather one would sometimes have a box be placed and be sized such that it "is just large enough to contain this, that, and that point". This situation typically arises when a picture has been drawn and, afterwards, parts of the picture are supposed to be encircled or highlighted.

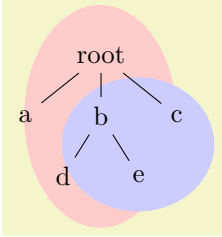
In this situation the `fit` option from the `fit` library is useful, see Section 54 for the details. The idea is that you may give the `fit` option to a node. The `fit` option expects a list of coordinates (one after the other without commas) as its parameter. The effect will be that the node's text area has exactly the necessary size so that it contains all the given coordinates. Here is an example:




```
\usetikzlibrary {fit,shapes.geometric}
\begin{tikzpicture}[level distance=8mm]
  \node (root) {root}
    child { node (a) {a} }
    child { node (b) {b}
      child { node (d) {d} }
      child { node (e) {e} } }
    child { node (c) {c} };

  \node[draw=red,inner sep=0pt,thick,ellipse,fit=(root) (b) (d) (e)] {};
  \node[draw=blue,inner sep=0pt,thick,ellipse,fit=(b) (c) (e)] {};
\end{tikzpicture}
```

If you want to fill the fitted node you will usually have to place it on a background layer.




```
\usetikzlibrary {backgrounds,fit,shapes.geometric}
\begin{tikzpicture}[level distance=8mm]
  \node (root) {root}
    child { node (a) {a} }
    child { node (b) {b}
      child { node (d) {d} }
      child { node (e) {e} } }
    child { node (c) {c} };

  \begin{scope}[on background layer]
    \node[fill=red!20,inner sep=0pt,ellipse,fit=(root) (b) (d) (e)] {};
    \node[fill=blue!20,inner sep=0pt,ellipse,fit=(b) (c) (e)] {};
  \end{scope}
\end{tikzpicture}
```

17.7 Transformations

It is possible to transform nodes, but, by default, transformations do not apply to nodes. The reason is that you usually do *not* want your text to be scaled or rotated even if the main graphic is transformed. Scaling text is evil, rotating slightly less so.

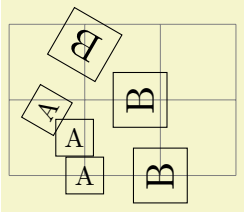
However, sometimes you *do* wish to transform a node, for example, it certainly sometimes makes sense to rotate a node by 90 degrees. There are two ways to achieve this:

1. You can use the following option:

   `/tikz/transform shape` (no value)

   Causes the current "external" transformation matrix to be applied to the shape. For example, if you said `\tikz[scale=3]` and then say `node[transform shape] {X}`, you will get a "huge" X in your graphic.

2. You can give transformation options *inside* the option list of the node. *These* transformations always apply to the node.

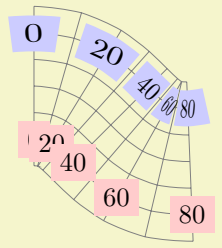



```
\usepgfmodule {nonlineartransformations}\usetikzlibrary {curvilinear}
\begin{tikzpicture}[every node/.style={draw}]
  \draw[help lines](0,0) grid (3,2);
  \draw            (1,0) node{A}
                   (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=30] (1,0) node{A}
                   (2,0) node[rotate=90,scale=1.5] {B};
  \draw[rotate=60] (1,0) node[transform shape] {A}
                   (2,0) node[transform shape,rotate=90,scale=1.5] {B};
\end{tikzpicture}
```

Even though Ti*k*Z currently does not allow you to configure so-called *nonlinear transformations,* see Section 108.4, there is an option that influences how nodes are transformed when nonlinear transformations are in force:

`/tikz/transform shape nonlinear`=⟨*true or false*⟩ (no default, initially `false`)

When set to true, Ti*k*Z will try to apply any current nonlinear transformation also to nodes. Typically, for the text in nodes this is not possible in general, in such cases a linear approximation of the nonlinear transformation is used. For more details, see Section 108.4.




```
\usepgfmodule {nonlineartransformations}\usetikzlibrary {curvilinear}
\begin{tikzpicture}
   % Install a nonlinear transformation:
   \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{10mm}{20mm}}
      {\pgfpoint{10mm}{10mm}}
      {\pgfpoint{20mm}{10mm}}
   \pgftransformnonlinear{\pgfpointcurvilinearbezierorthogonal\pgf@x\pgf@y}%

   % Draw something:
   \draw [help lines] (0,-30pt) grid [step=10pt] (80pt,30pt);

   \foreach \x in {0,20,...,80}
     \node [fill=red!20]  at (\x pt, -20pt) {\x};

   \foreach \x in {0,20,...,80}
     \node [fill=blue!20, transform shape nonlinear] at (\x pt, 20pt) {\x};
\end{tikzpicture}
```

17.8 Placing Nodes on a Line or Curve Explicitly

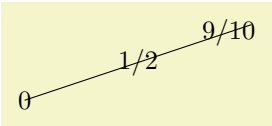
Until now, we always placed nodes on a coordinate that is mentioned in the path. Often, however, we wish to place nodes on "the middle" of a line and we do not wish to compute these coordinates "by hand". To facilitate such placements, Ti*k*Z allows you to specify that a certain node should be somewhere "on" a line. There are two ways of specifying this: Either explicitly by using the `pos` option or implicitly by placing the node "inside" a path operation. These two ways are described in the following.

`/tikz/pos`=⟨*fraction*⟩ (no default)

When this option is given, the node is not anchored on the last coordinate. Rather, it is anchored on some point on the line from the previous coordinate to the current point. The ⟨*fraction*⟩ dictates how "far" on the line the point should be. A ⟨*fraction*⟩ of 0 is the previous coordinate, 1 is the current one, everything else is in between. In particular, 0.5 is the middle.

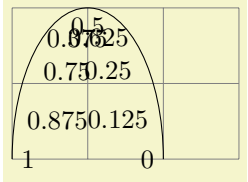
Now, what is "the previous line"? This depends on the previous path construction operation.

In the simplest case, the previous path operation was a "line-to" operation, that is, a `--`⟨*coordinate*⟩ operation:




```
\tikz \draw (0,0) -- (3,1)
    node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

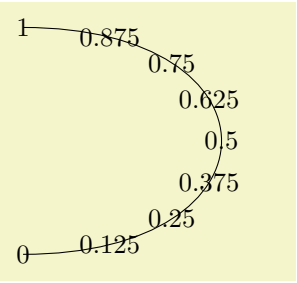
For the `arc` operation, the position is simply the corresponding position on the arc:




```
\tikz {
  \draw [help lines] (0,0) grid (3,2);
  \draw (2,0) arc [x radius=1, y radius=2, start angle=0, end angle=180]
              node foreach \t in {0,0.125,...,1} [pos=\t,auto] {\t};
}
```

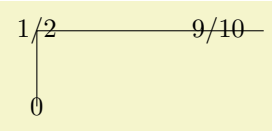
The next case is the curve-to operation (the `..` operation). In this case, the "middle" of the curve, that is, the position `0.5` is not necessarily the point at the exact half distance on the line. Rather, it is some

point at “time” 0.5 of a point traveling from the start of the curve, where it is at time 0, to the end of the curve, which it reaches at time 0.5. The “speed” of the point depends on the length of the support vectors (the vectors that connect the start and end points to the control points). The exact math is a bit complicated (depending on your point of view, of course); you may wish to consult a good book on computer graphics and Bézier curves if you are intrigued.

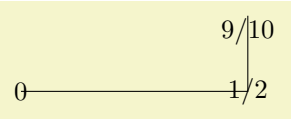



```
\tikz \draw (0,0) .. controls +(right:3.5cm) and +(right:3.5cm) .. (0,3)
  node foreach \p in {0,0.125,...,1} [pos=\p]{\p};
```

Another interesting case are the horizontal/vertical line-to operations `|-` and `-|`. For them, the position (or time) `0.5` is exactly the corner point.




```
\tikz \draw (0,0) |- (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```



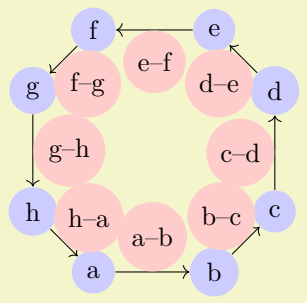

```
\tikz \draw (0,0) -| (3,1)
  node[pos=0]{0} node[pos=0.5]{1/2} node[pos=0.9]{9/10};
```

For all other path construction operations, *the position placement does not work*, currently.

`/tikz/auto=`⟨*direction*⟩ (default is scope’s setting)

This option causes an anchor position to be calculated automatically according to the following rule. Consider a line between two points. If the ⟨*direction*⟩ is `left`, then the anchor is chosen such that the node is to the left of this line. If the ⟨*direction*⟩ is `right`, then the node is to the right of this line. Leaving out ⟨*direction*⟩ causes automatic placement to be enabled with the last value of `left` or `right` used. A ⟨*direction*⟩ of `false` disables automatic placement. This happens also whenever an anchor is given explicitly by the `anchor` option or by one of the `above`, `below`, etc. options.

This option only has an effect for nodes that are placed on lines or curves.

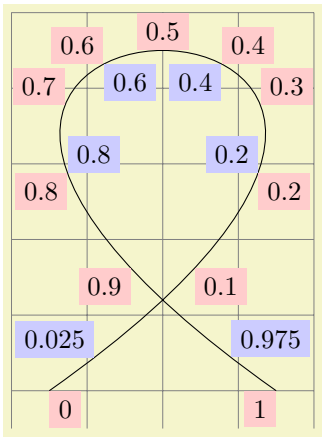



```
\begin{tikzpicture}
  [scale=.8,auto=left,every node/.style={circle,fill=blue!20}]
  \node (a) at (-1,-2) {a};
  \node (b) at ( 1,-2) {b};
  \node (c) at ( 2,-1) {c};
  \node (d) at ( 2, 1) {d};
  \node (e) at ( 1, 2) {e};
  \node (f) at (-1, 2) {f};
  \node (g) at (-2, 1) {g};
  \node (h) at (-2,-1) {h};

  \foreach \from/\to in {a/b,b/c,c/d,d/e,e/f,f/g,g/h,h/a}
    \draw [->] (\from) -- (\to)
               node[midway,fill=red!20] {\from--\to};
\end{tikzpicture}
```

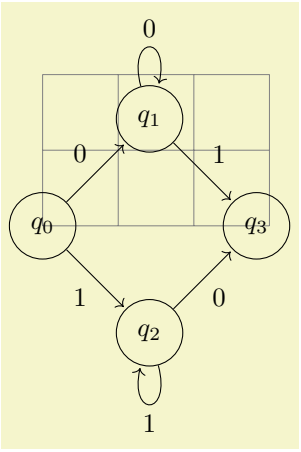
`/tikz/swap` (no value)

This option exchanges the roles of `left` and `right` in automatic placement. That is, if `left` is the current `auto` placement, `right` is set instead and the other way round.




```
\usetikzlibrary {automata}
\begin{tikzpicture}[auto]
  \draw[help lines,use as bounding box] (0,-.5) grid (4,5);

  \draw (0.5,0) .. controls (9,6) and (-5,6) .. (3.5,0)
    node foreach \pos in {0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1}
         [pos=\pos,swap,fill=red!20] {\pos}
    node foreach \pos in {0.025,0.2,0.4,0.6,0.8,0.975}
         [pos=\pos,fill=blue!20] {\pos};
\end{tikzpicture}
```




```
\usetikzlibrary {automata}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,auto]
  \draw[help lines] (0,0) grid (3,2);

  \node[state] (q_0)                      {$q_0$};
  \node[state] (q_1) [above right of=q_0] {$q_1$};
  \node[state] (q_2) [below right of=q_0] {$q_2$};
  \node[state] (q_3) [below right of=q_1] {$q_3$};

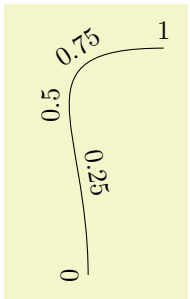
  \path[->] (q_0) edge              node        {0} (q_1)
                  edge              node [swap] {1} (q_2)
            (q_1) edge              node        {1} (q_3)
                  edge [loop above] node        {0} ()
            (q_2) edge              node [swap] {0} (q_3)
                  edge [loop below] node        {1} ();
\end{tikzpicture}
```

/tikz/' (no value)

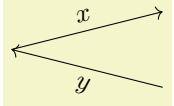
This is a very short alias for `swap`.

/tikz/sloped (no value)

This option causes the node to be rotated such that a horizontal line becomes a tangent to the curve. The rotation is normally done in such a way that text is never "upside down". To get upside-down text, use can use `[rotate=180]` or `[allow upside down]`, see below.



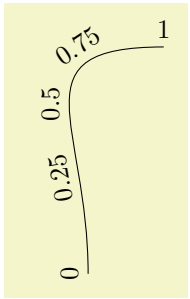

```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```



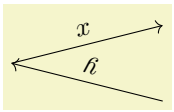

```
\begin{tikzpicture}[->]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

/tikz/allow upside down=⟨*boolean*⟩ (default `true`, initially `false`)

If set to `true`, Ti*k*Z will not "righten" upside down text.




```
\tikz [allow upside down]
  \draw (0,0) .. controls +(up:2cm) and +(left:2cm) .. (1,3)
    node foreach \p in {0,0.25,...,1} [sloped,above,pos=\p]{\p};
```

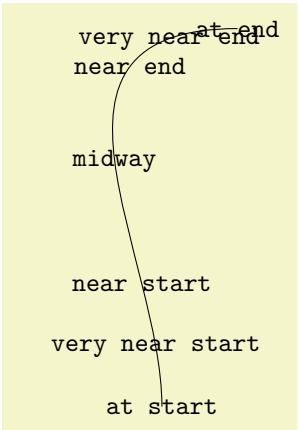



```
\begin{tikzpicture}[->,allow upside down]
  \draw (0,0)   -- (2,0.5) node[midway,sloped,above] {$x$};
  \draw (2,-.5) -- (0,0)   node[midway,sloped,below] {$y$};
\end{tikzpicture}
```

There exist styles for specifying positions a bit less "technically":

/tikz/midway (style, no value)

This has the same effect as `pos=0.5`.




```
\tikz \draw (0,0) .. controls +(up:2cm) and +(left:3cm) .. (1,5)
      node[at end]          {\texttt{at end}}
      node[very near end]   {\texttt{very near end}}
      node[near end]        {\texttt{near end}}
      node[midway]          {\texttt{midway}}
      node[near start]      {\texttt{near start}}
      node[very near start] {\texttt{very near start}}
      node[at start]        {\texttt{at start}};
```

/tikz/near start (style, no value)

Set to `pos=0.25`.

/tikz/near end (style, no value)

Set to `pos=0.75`.

/tikz/very near start (style, no value)

Set to `pos=0.125`.

/tikz/very near end (style, no value)

Set to `pos=0.875`.

/tikz/at start (style, no value)

Set to `pos=0`.

/tikz/at end (style, no value)

Set to `pos=1`.

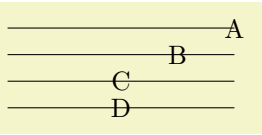
17.9 Placing Nodes on a Line or Curve Implicitly

When you wish to place a node on the line `(0,0) -- (1,1)`, it is natural to specify the node not following the `(1,1)`, but "somewhere in the middle". This is, indeed, possible and you can write `(0,0) -- node{a} (1,1)` to place a node midway between `(0,0)` and `(1,1)`.

What happens is the following: The syntax of the line-to path operation is actually `-- node`⟨*node specification*⟩⟨*coordinate*⟩. (It is even possible to give multiple nodes in this way.) When the optional `node` is encountered, that is, when the `--` is directly followed by `node`, then the specification(s) are read and "stored away". Then, after the ⟨*coordinate*⟩ has finally been reached, they are inserted again, but with the `pos` option set.

There are two things to note about this: When a node specification is "stored", its catcodes become fixed. This means that you cannot use overly complicated verbatim text in them. If you really need, say, a verbatim text, you will have to put it in a normal node following the coordinate and add the `pos` option.

Second, which `pos` is chosen for the node? The position is inherited from the surrounding scope. However, this holds only for nodes specified in this implicit way. Thus, if you add the option `[near end]` to a scope, this does not mean that *all* nodes given in this scope will be put on near the end of lines. Only the nodes for which an implicit `pos` is added will be placed near the end. Typically, this is what you want. Here are some examples that should make this clearer:




```
\begin{tikzpicture}[near end]
  \draw (0cm,4em) -- (3cm,4em) node{A};
  \draw (0cm,3em) --           node{B}          (3cm,3em);
  \draw (0cm,2em) --           node[midway] {C} (3cm,2em);
  \draw (0cm,1em) -- (3cm,1em) node[midway] {D} ;
\end{tikzpicture}
```

Like the line-to operation, the curve-to operation `..` also allows you to specify nodes "inside" the operation. After both the first `..` and also after the second `..` you can place node specifications. Like for the `--` operation, these will be collected and then reinserted after the operation with the `pos` option set.

17.10 The Label and Pin Options

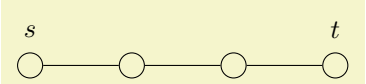
17.10.1 Overview

In addition to the `node` path operation, the two options `label` and `pin` can be used to "add a node next to another node". As an example, suppose we want to draw a graph in which the nodes are small circles:



```
\usetikzlibrary {positioning}
\tikz [circle] {
  \node [draw] (s) {};
  \node [draw] (a) [right=of s] {} edge (s);
  \node [draw] (b) [right=of a] {} edge (a);
  \node [draw] (t) [right=of b] {} edge (b);
}
```

Now, in the above example, suppose we wish to indicate that the first node is the start node and the last node is the target node. We could write `\node (s) {$s$};`, but this would enlarge the first node. Rather, we want the "s" to be placed next to the node. For this, we need to create *another* node, but next to the existing node. The `label` and `pin` option allow us to do exactly this without having to use the cumbersome `node` syntax:




```
\usetikzlibrary {positioning}
\tikz [circle] {
  \node [draw] (s) [label=$s$]  {};
  \node [draw] (a) [right=of s] {} edge (s);
  \node [draw] (b) [right=of a] {} edge (a);
  \node [draw] (t) [right=of b, label=$t$] {} edge (b);
}
```

17.10.2 The Label Option

`/tikz/label=[`⟨*options*⟩`]`⟨*angle*⟩`:`⟨*text*⟩ (no default)

When this option is given to a `node` operation, it causes *another* node to be added to the path after the current node has been finished. This extra node will have the text ⟨*text*⟩. It is placed, in principle, in the direction ⟨*angle*⟩ relative to the main node, but the exact rules are a bit complex. Suppose the `node` currently under construction is called `main node` and let us call the label node `label node`. Then the following happens:

1. The ⟨*angle*⟩ is used to determine a position on the border of the `main node`. If the ⟨*angle*⟩ is missing, the value of the following key is used instead:

   `/tikz/label position=`⟨*angle*⟩ (no default, initially `above`)

   Sets the default position for labels.

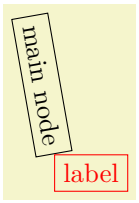
   The ⟨*angle*⟩ determines the position on the border of the shape in two different ways. Normally, the border position is given by `main node.`⟨*angle*⟩. This means that the ⟨*angle*⟩ can either be a number like `0` or `-340`, but it can also be an anchor like `north`. Additionally, the special angles `above`, `below`, `left`, `right`, `above left`, and so on are automatically replaced by the corresponding angles `90`, `270`, `180`, `0`, `135`, and so on.

   A special case arises when the following key is set:

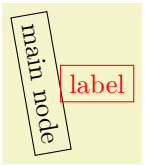
   `/tikz/absolute=`⟨*true or false*⟩ (default `true`)

   When this key is set, the ⟨*angle*⟩ is interpreted differently: We still use a point on the border of the `main node`, but the angle is measured "absolutely", that is, an angle of `0` refers to the point on the border that lies on a straight line from the `main node`'s center to the right (relative to the paper, not relative to the local coordinate system of either the node or the scope).

   The difference can be seen in the following example:



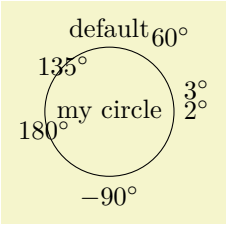

```
\tikz [rotate=-80,every label/.style={draw,red}]
  \node [transform shape,rectangle,draw,label=right:label] {main node};
```




```
\tikz [rotate=-80,every label/.style={draw,red},absolute]
  \node [transform shape,rectangle,draw,label=right:label] {main node};
```

2. Then, an anchor point for the `label node` is computed. It is determined in such a way that the `label node` will "face away" from the border of the `main node`. The anchor that is chosen depends on the position of the border point that is chosen and its position relative to the center of the `main node` and on whether the `transform shape` option is set. In detail, when the computed border point is at 0°, the anchor `west` will be used. Similarly, when the border point is at 90°, the anchor `south` will be used, and so on for 180° and 270°.

   For angles between these "major" angles, like 30° or 110°, combined anchors, like `south west` for 30° or `south east` for 110°, are used. However, for angles close to the major angles, (differing by up to 2° from the major angle), the anchor for the major angle is used. Thus, a label at a border point for 2° will have the anchor `west`, while a label for 3° will have the anchor `south west`, resulting in a "jump" of the anchor. You can set the anchor "by hand" using the `anchor` key or indirect keys like `left`.



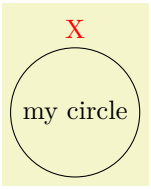

```
\tikz
  \node [circle, draw,
         label=default,
         label=60:$60^\circ$,
         label=below:$-90^\circ$,
         label=3:$3^\circ$,
         label=2:$2^\circ$,
         label={[below]180:$180^\circ$},
         label={[centered]135:$135^\circ$}] {my circle};
```

3. One ⟨*angle*⟩ is special: If you set the ⟨*angle*⟩ to `center`, then the label will be placed on the center of the main node. This is mainly useful for adding a label text to an existing node, especially if it has been rotated.

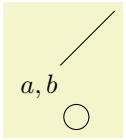



```
\tikz \node [transform shape,rotate=90,
             rectangle,draw,label={[red]center:R}] {main node};
```

You can pass ⟨*options*⟩ to the node `label node`. For this, you provide the options in square brackets before the ⟨*angle*⟩. If you do so, you need to add braces around the whole argument of the `label` option and this is also the case if you have brackets or commas or semicolons or anything special in the ⟨*text*⟩.




```
\tikz \node [circle,draw,label={[red]above:X}] {my circle};
```



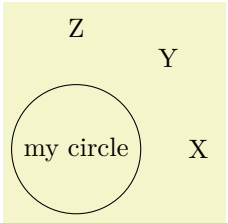

```
\begin{tikzpicture}
  \node [circle,draw,label={[name=label node]above left:$a,b$}] {};
  \draw (label node) -- +(1,1);
\end{tikzpicture}
```

If you provide multiple `label` options, then multiple extra label nodes are added in the order they are given.

The following styles influence how labels are drawn:

`/tikz/label distance=`⟨*distance*⟩ (no default, initially `0pt`)

The ⟨*distance*⟩ is additionally inserted between the main node and the label node.




```
\tikz[label distance=5mm]
  \node [circle,draw,label=right:X,
                     label=above right:Y,
                     label=above:Z]       {my circle};
```

`/tikz/every label` (style, initially empty)

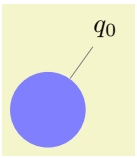
This style is used in every node created by the `label` option. The default is `draw=none,fill=none`.

See Section 17.10.4 for an easier syntax for specifying nodes.

17.10.3 The Pin Option

/tikz/pin=[⟨*options*⟩]⟨*angle*⟩:⟨*text*⟩ (no default)

This option is quite similar to the `label` option, but there is one difference: In addition to adding an extra node to the picture, it also adds an edge from this node to the main node. This causes the node to look like a pin that has been added to the main node:

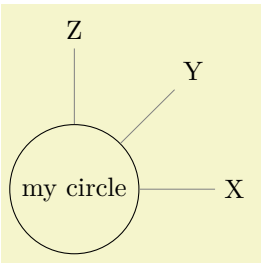



```
\tikz \node [circle,fill=blue!50,minimum size=1cm,pin=60:$q_0$] {};
```

The meaning of the ⟨*options*⟩ and the ⟨*angle*⟩ and the ⟨*text*⟩ is exactly the same as for the `node` option. Only, the options and styles the influence the way pins look are different:

/tikz/pin distance=⟨*distance*⟩ (no default, initially `3ex`)

This ⟨*distance*⟩ is used instead of the `label distance` for the distance between the main node and the label node.




```
\tikz[pin distance=1cm]
  \node [circle,draw,pin=right:X,
                     pin=above right:Y,
                     pin=above:Z]       {my circle};
```

/tikz/every pin (style, initially `draw=none,fill=none`)

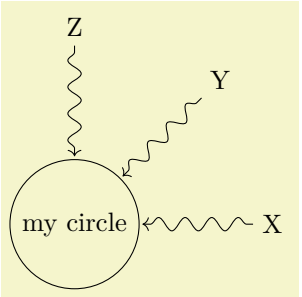
This style is used in every node created by the `pin` option.

/tikz/pin position=⟨*angle*⟩ (no default, initially `above`)

The default pin position. Works like `label position`.

/tikz/every pin edge (style, initially `help lines`)

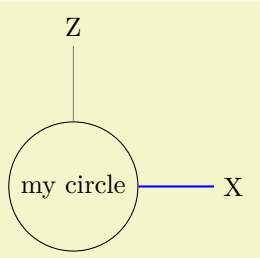
This style is used in every edge created by the `pin` options.



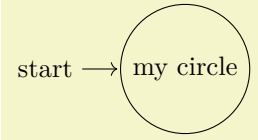

```
\usetikzlibrary {decorations.pathmorphing}
\tikz [pin distance=15mm,
       every pin edge/.style={<-,shorten <=1pt,decorate,
                              decoration={snake,pre length=4pt}}]
  \node [circle,draw,pin=right:X,
                     pin=above right:Y,
                     pin=above:Z]       {my circle};
```

/tikz/pin edge=⟨*options*⟩ (no default, initially empty)

This option can be used to set the options that are to be used in the edge created by the `pin` option.




```
\tikz[pin distance=10mm]
  \node [circle,draw,pin={[pin edge={blue,thick}]right:X},
                         pin=above:Z]        {my circle};
```




```
\tikz [every pin edge/.style={},
       initial/.style={pin={[pin distance=5mm,
                             pin edge={<-,shorten <=1pt}]left:start}}]
  \node [circle,draw,initial] {my circle};
```

17.10.4 The Quotes Syntax

The `label` and `pin` options provide a syntax for creating nodes next to existing nodes, but this syntax is often a bit too verbose. By including the following library, you get access to an even more concise syntax:

TikZ Library quotes

```
\usetikzlibrary{quotes} % LaTeX and plain TeX
\usetikzlibrary[quotes] % ConTeXt
```

Enables the quotes syntax for labels, pins, edge nodes, and pic texts.

Let us start with the basics of what this library does: Once loaded, inside the options of a `node` command, instead of the usual ⟨*key*⟩=⟨*value*⟩ pairs, you may also provide strings of the following form (the actual syntax is slightly more general, see the detailed descriptions later on):

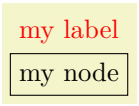
"⟨*text*⟩"⟨*options*⟩

The ⟨*options*⟩ must be surrounded in curly braces when they contain a comma, otherwise the curly braces are optional. The ⟨*options*⟩ may be preceded by an optional space.

When a ⟨*string*⟩ of the above form is encountered inside the options of a `node`, then it is internally transformed to

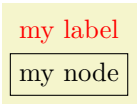
`label={[`⟨*options*⟩`]`⟨*text*⟩`}`

Let us have a look at an example:



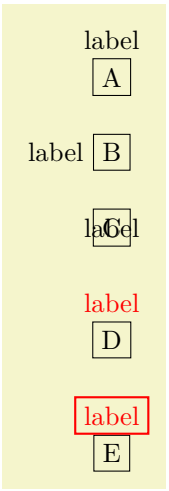

```
\usetikzlibrary {quotes}
\tikz \node ["my label" red, draw] {my node};
```

The above has the same effect as the following:




```
\tikz \node [label={[red]my label}, draw] {my node};
```

Here are further examples, one where no ⟨*options*⟩ are added to the `label`, one where a position is specified, and examples with more complicated options in curly braces:




```
\usetikzlibrary {quotes}
\begin{tikzpicture}
  \matrix [row sep=5mm] {
    \node [draw, "label"]                  {A}; \\
    \node [draw, "label" left]             {B}; \\
    \node [draw, "label" centered]         {C}; \\
    \node [draw, "label" color=red]        {D}; \\
    \node [draw, "label" {red,draw,thick}] {E}; \\
  };
\end{tikzpicture}
```

Let us now have a more detailed look at what commands this library provides:

`/tikz/quotes mean label` (no value)

When this option is used (which is the default when this library is loaded), then, as described above, inside the options of a node a special syntax check is done.

The syntax. For each string in the list of options it is tested whether it starts with a quotation mark (note that this will never happen for normal keys since the normal keys of TikZ do not start with quotation marks). When this happens, the ⟨*string*⟩ should not be a key–value pair, but, rather, must have the form:

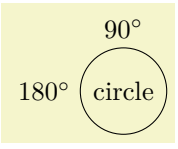
"⟨*text*⟩"'⟨*options*⟩

(We will discuss the optional apostrophe in a moment. It is not really important for the current option, but only for edge labels, which are discussed later).

Transformation to a label option. When a ⟨*string*⟩ has the above form, it is treated (almost) as if you had written

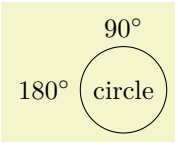
`label={[`⟨*options*⟩`]`⟨*text*⟩`}`

instead. The "almost" refers to the following additional feature: In reality, before the ⟨*options*⟩ are executed inside the `label` command, the direction keys `above`, `left`, `below` `right` and so on are redefined so that `above` is a shorthand for `label position=90` and similarly for the other keys. The net effect is that in order to specify the position of the ⟨*text*⟩ relative to the main node you can just put something like `left` or `above right` inside the ⟨*options*⟩:




```
\usetikzlibrary {quotes}
\tikz
  \node ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle};
```

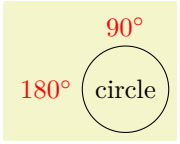
Alternatively, you can also use ⟨*direction*⟩`:`⟨*actual text*⟩ as your ⟨*text*⟩. This works since the `label` command allows you to specify a direction at the beginning when it is separated by a colon:




```
\usetikzlibrary {quotes}
\tikz
  \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

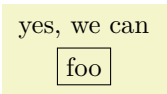
Arguably, placing `above` or `left` behind the ⟨*text*⟩ seems more natural than having it inside the ⟨*text*⟩. In addition to the above, before the ⟨*options*⟩ are executed, the following style is also executed:

`/tikz/every label quotes` (style, no value)



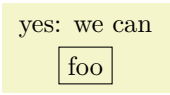

```
\usetikzlibrary {quotes}
\tikz [every label quotes/.style=red]
  \node ["90:$90^\circ$", "left:$180^\circ$", circle, draw] {circle};
```

Handling commas and colons inside the text. The ⟨*text*⟩ may not contain a comma, unless it is inside curly braces. The reason is that the key handler separates the total options of a `node` along the commas it finds. So, in order to have text containing a comma, just add curly braces around either the comma or just around the whole ⟨*text*⟩:




```
\usetikzlibrary {quotes}
\tikz \node ["{yes, we can}", draw] {foo};
```

The same is true for a colon, only in this case you may need to surround specifically the colon by curly braces to stop the `label` option from interpreting everything before the colon as a direction:



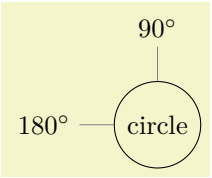

```
\usetikzlibrary {quotes}
\tikz \node ["yes{:} we can", draw] {foo};
```

The optional apostrophe. Following the closing quotation marks in a ⟨*string*⟩ there may (but need not) be a single quotation mark (an apostrophe), possibly surrounded by whitespaces. If it is present, it is simply added to the ⟨*options*⟩ as another option (and, indeed, a single apostrophe is a legal option in Ti*k*Z, it is a shorthand for `swap`):

String	has the same effect as
`"foo"'`	`"foo" {'}`
`"foo"' red`	`"foo" {',red}`
`"foo"'{red}`	`"foo" {',red}`
`"foo"{',red}`	`"foo" {',red}`
`"foo"{red,'}`	`"foo" {red,'}`
`"foo"{'red}`	`"foo" {'red}` (illegal; there is no key `'red`)
`"foo" red'`	`"foo" {red'}` (illegal; there is no key `red'`)

`/tikz/quotes mean pin` (no value)

This option has exactly the same effect as `quotes mean label`, only instead of transforming quoted text to the `label` option, they get transformed to the `pin` option:




```
\usetikzlibrary {quotes}
\tikz [quotes mean pin]
  \node ["$90^\circ$" above, "$180^\circ$" left, circle, draw] {circle};
```

Instead of `every label quotes`, the following style is executed with each such pin:

`/tikz/every pin quotes` (style, no value)

If instead of `labels` or `pins` you would like quoted strings to be interpreted in a different manner, you can also define your own handlers:

`/tikz/node quotes mean=`⟨*replacement*⟩ (no default)

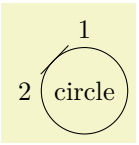
This key allows you to define your own handler for quotes options. Inside the options of a `node`, whenever a key–value pair with the syntax

"⟨*text*⟩"'⟨*options*⟩

is encountered, the following happens: The above string gets replaced by ⟨*replacement*⟩ where inside the ⟨*replacement*⟩ the parameter `#1` is ⟨*text*⟩ and `#2` is ⟨*options*⟩. If the apostrophe is present (see also the discussion of `quotes mean label`), the ⟨*options*⟩ start with `',`.

The ⟨*replacement*⟩ is then parsed normally as options (using `\pgfkeys`).

Here is an example, where the quotes are used to define labels that are automatically named according to the `text`:




```
\usetikzlibrary {quotes}
\tikzset{node quotes mean={label={[#2,name={#1}]#1}}}

\tikz {
  \node ["1", "2" label position=left, circle, draw] {circle};
  \draw (1) -- (2);
}
```

Some further options provided by the `quotes` library concern labels next to edges rather than nodes and they are described in Section 17.12.2.

17.11 Connecting Nodes: Using Nodes as Coordinates

Once you have defined a node and given it a name, you can use this name to reference it. This can be done in two ways, see also Section 13.2.3. Suppose you have said `\path(0,0) node(x) {Hello World!};` in order to define a node named `x`.

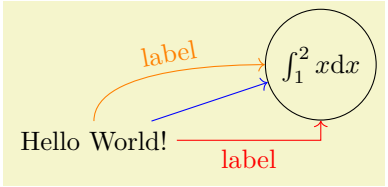
1. Once the node `x` has been defined, you can use `(x.`⟨*anchor*⟩`)` wherever you would normally use a normal coordinate. This will yield the position at which the given ⟨*anchor*⟩ is in the picture. Note that transformations do not apply to this coordinate, that is, `(x.north)` will be the northern anchor of `x` even if you have said `scale=3` or `xshift=4cm`. This is usually what you would expect.

2. You can also just use `(x)` as a coordinate. In most cases, this gives the same coordinate as `(x.center)`. Indeed, if the `shape` of `x` is `coordinate`, then `(x)` and `(x.center)` have exactly the same effect.

   However, for most other shapes, some path construction operations like `--` try to be "clever" when they are asked to draw a line from such a coordinate or to such a coordinate. When you say `(x)--(1,1)`, the `--` path operation will not draw a line from the center of `x`, but *from the border* of `x` in the direction going towards `(1,1)`. Likewise, `(1,1)--(x)` will also have the line end on the border in the direction coming from `(1,1)`.

   If the specified coordinate is almost identical to the node center, for example `(x)--(0,0)`, no line will be drawn and a warning message will be printed.

   In addition to `--`, the curve-to path operation `..` and the path operations `-|` and `|-` will also handle nodes without anchors correctly. Here is an example, see also Section 13.2.3:




```
\begin{tikzpicture}
  \path (0,0) node             (x) {Hello World!}
        (3,1) node[circle,draw](y) {$\int_1^2 x \mathrm d x$};

  \draw[->,blue]   (x) -- (y);
  \draw[->,red]    (x) -| node[near start,below] {label} (y);
  \draw[->,orange] (x) .. controls +(up:1cm) and +(left:1cm) .. node[above,sloped] {label} (y);
\end{tikzpicture}
```

17.12 Connecting Nodes: Using the Edge Operation

17.12.1 Basic Syntax of the Edge Operation

The `edge` operation works like a `to` operation that is added after the main path has been drawn, much like a node is added after the main path has been drawn. This allows each `edge` to have a different appearance. As the `node` operation, an `edge` temporarily suspends the construction of the current path and a new path p is constructed. This new path p will be drawn after the main path has been drawn. Note that p can be totally different from the main path with respect to its options. Also note that if there are several `edge` and/or `node` operations in the main path, each creates its own path(s) and they are drawn in the order that they are encountered on the main path.

`\path` … `edge[`⟨*options*⟩`]` ⟨*nodes*⟩ `(`⟨*coordinate*⟩`)` … `;`

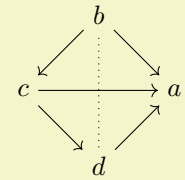
The effect of the `edge` operation is that after the main path the following path is added to the picture:

`\path[every edge,`⟨*options*⟩`] (\tikztostart)` ⟨*path*⟩`;`

Here, ⟨*path*⟩ is the `to path`. Note that, unlike the path added by the `to` operation, the `(\tikztostart)` is added before the ⟨*path*⟩ (which is unnecessary for the `to` operation, since this coordinate is already part of the main path).

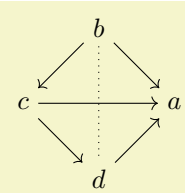
The `\tikztostart` is the last coordinate on the path just before the `edge` operation, just as for the `node` or `to` operations. However, there is one exception to this rule: If the `edge` operation is directly preceded by a `node` operation, then this just-declared node is the start coordinate (and not, as would normally be the case, the coordinate where this just-declared node is placed – a small, but subtle difference). In this regard, `edge` differs from both `node` and `to`.

If there are several `edge` operations in a row, the start coordinate is the same for all of them as their target coordinates are not, after all, part of the main path. The start coordinate is, thus, the coordinate preceding the first `edge` operation. This is similar to nodes insofar as the `edge` operation does not modify the current path at all. In particular, it does not change the last coordinate visited, see the following example:




```
\begin{tikzpicture}
  \node (a) at   (0:1) {$a$};
  \node (b) at  (90:1) {$b$} edge [->]      (a);
  \node (c) at (180:1) {$c$} edge [->]      (a)
                             edge [<-]      (b);
  \node (d) at (270:1) {$d$} edge [->]      (a)
                             edge [dotted]  (b)
                             edge [<-]      (c);
\end{tikzpicture}
```

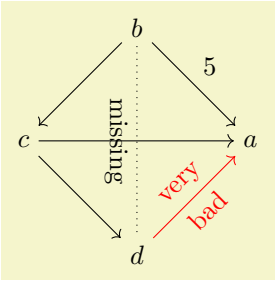
A different way of specifying the above graph using the `edge` operation is the following:




```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
        (\name) at (\angle:1) {$\name$};

  \path[->] (b) edge (a)
                edge (c)
                edge [-,dotted] (d)
            (c) edge (a)
                edge (d)
            (d) edge (a);
\end{tikzpicture}
```

As can be seen, the path of the `edge` operation inherits the options from the main path, but you can locally overrule them.



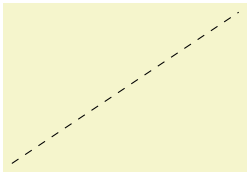

```
\begin{tikzpicture}
  \node foreach \name/\angle in {a/0,b/90,c/180,d/270}
        (\name) at (\angle:1.5) {$\name$};

  \path[->] (b) edge            node[above right]  {$5$}     (a)
                edge                                         (c)
                edge [-,dotted] node[below,sloped] {missing} (d)
            (c) edge                                         (a)
                edge                                         (d)
            (d) edge [red]      node[above,sloped] {very}
                                node[below,sloped] {bad}     (a);
\end{tikzpicture}
```

Instead of `every to`, the style `every edge` is installed at the beginning of the main path.

`/tikz/every edge` (style, initially `draw`)

Executed for each `edge`.



```
\begin{tikzpicture}[every edge/.style={draw,dashed}]
  \path (0,0) edge (3,2);
\end{tikzpicture}
```

17.12.2 Nodes on Edges: Quotes Syntax

The standard way of specifying nodes that are placed "on" an edge (or on a to-path; all of the following is also true for to–paths) is to put node specifications after the `edge` keyword, but before the target coordinate. Another way is to use the `edge node` option and its friends. Yet another way is to use the quotes syntax.

The syntax is essentially the same as for labels added to nodes as described in Section 17.10.4 and you also need to load the `quotes` library.

In detail, when the `quotes` library is loaded, each time a key–value pair in a list of options passed to an `edge` or a `to` path command starts with `"`, the key–value pair must actually be a string of the following form:

`"`⟨*text*⟩`"'`⟨*options*⟩

This string is transformed into the following:

`edge node=node [every edge quotes,`⟨*options*⟩`]{`⟨*text*⟩`}`

As described in Section 17.10.4, the apostrophe becomes part of the ⟨*options*⟩, when present.

The following style is important for the placement of the labels:

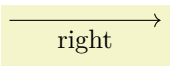
`/tikz/every edge quotes` (style, initially `auto`)

This style is `auto` by default, which causes labels specified using the quotes-syntax to be placed next to the edges. Unless the setting of `auto` has been changed, they will be placed to the left.



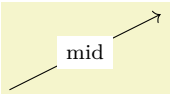

```
\usetikzlibrary {quotes}
\tikz \draw (0,0) edge ["left", ->] (2,0);
```

In order to place all labels to the right by default, change this style to `auto=right`:



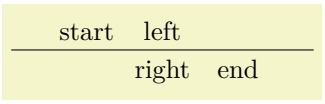

```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={auto=right}]
  \draw (0,0) edge ["right", ->] (2,0);
```

To place all nodes "on" the edge, just make this style empty (and, possibly, make your labels opaque):



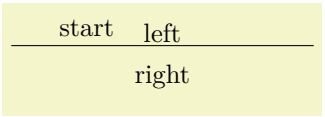

```
\usetikzlibrary {quotes}
\tikz [every edge quotes/.style={fill=white,font=\footnotesize}]
  \draw (0,0) edge ["mid", ->] (2,1);
```

You may often wish to place some edge nodes to the right of edges and some to the left. For this, the special treatment of the apostrophe is particularly convenient: Recall that in Ti*k*Z there is an option just called `'`, which is a shorthand for `swap`. Now, following the closing quotation mark come the options of an edge node. Thus, if the closing quotation mark is followed by an apostrophe, the `swap` option will be added to the edge label, causing it is be placed on the other side. Because of the special treatment, you can even add another option like `near end` after the apostrophe without having to add curly braces and commas:




```
\usetikzlibrary {quotes}
\tikz
  \draw (0,0) edge ["left", "right"',
                    "start" near start,
                    "end"' near end] (4,0);
```

In order to modify the distance between the edge labels and the edge, you should consider introducing some styles:




```
\usetikzlibrary {quotes}
\tikz [tight/.style={inner sep=1pt}, loose/.style={inner sep=.7em}]
  \draw (0,0) edge ["left"   tight,
                    "right"' loose,
                    "start"  near start] (4,0);
```

17.13 Referencing Nodes Outside the Current Picture

17.13.1 Referencing a Node in a Different Picture

It is possible (but not quite trivial) to reference nodes in pictures other than the current one. This means that you can create a picture and a node therein and, later, you can draw a line from some other position to this node.

To reference nodes in different pictures, proceed as follows:

1. You need to add the `remember picture` option to all pictures that contain nodes that you wish to reference and also to all pictures from which you wish to reference a node in another picture.
2. You need to add the `overlay` option to paths or to whole pictures that contain references to nodes in different pictures. (This option switches the computation of the bounding box off.)
3. You need to use a driver that supports picture remembering and you need to run TEX twice.

(For more details on what is going on behind the scenes, see Section 106.3.2.)

Let us have a look at the effect of these options.

`/tikz/remember picture`=⟨*boolean*⟩ (no default, initially `false`)

This option tells Ti*k*Z that it should attempt to remember the position of the current picture on the page. This attempt may fail depending on which backend driver is used. Also, even if remembering works, the position may only be available on a second run of TEX.

Provided that remembering works, you may consider saying

```
\tikzset{every picture/.append style={remember picture}}
```

to make Ti*k*Z remember all pictures. This will add one line in the `.aux` file for each picture in your document – which typically is not very much. Then, you do not have to worry about remembered pictures at all.

`/tikz/overlay`=⟨*boolean*⟩ (default `true`)

This option is mainly intended for use when nodes in other pictures are referenced, but you can also use it in other situations. The effect of this option is that everything within the current scope is not taken into consideration when the bounding box of the current picture is computed.

You need to specify this option on all paths (or at least on all parts of paths) that contain a reference to a node in another picture. The reason is that, otherwise, Ti*k*Z will attempt to make the current picture large enough to encompass *the node in the other picture*. However, on a second run of TEX this will create an even bigger picture, leading to larger and larger pictures. Unless you know what you are doing, I suggest specifying the `overlay` option with all pictures that contain references to other pictures.

Let us now have a look at a few examples. These examples work only if this document is processed with a driver that supports picture remembering.

Inside the current text we place two pictures, containing nodes named `n1` and `n2`, using

```
\tikz[remember picture] \node[circle,fill=red!50] (n1) {};
```

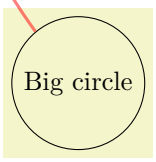
which yields , and

```
\tikz[remember picture] \node[fill=blue!50] (n2) {};
```

yielding the node . To connect these nodes, we create another picture using the `overlay` option and also the `remember picture` option.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw[->,very thick] (n1) -- (n2);
\end{tikzpicture}
```

Note that the last picture is seemingly empty. What happens is that it has zero size and contains an arrow that lies well outside its bounds. As a last example, we connect a node in another picture to the first two nodes. Here, we provide the `overlay` option only with the line that we do not wish to count as part of the picture.




```
\begin{tikzpicture}[remember picture]
  \node (c) [circle,draw] {Big circle};

  \draw [overlay,->,very thick,red,opacity=.5]
    (c) to[bend left] (n1) (n1) -| (n2);
\end{tikzpicture}
```

17.13.2 Referencing the Current Page Node – Absolute Positioning

There is a special node called `current page` that can be used to access the current page. It is a node of shape rectangle whose `south west` anchor is the lower left corner of the page and whose `north east` anchor is the upper right corner of the page. While this node is handled in a special way internally, you can reference it as if it were defined in some remembered picture other than the current one. Thus, by giving the `remember picture` and the `overlay` options to a picture, you can position nodes *absolutely* on a page.

The first example places some text in the lower left corner of the current page:

```
\begin{tikzpicture}[remember picture,overlay]
  \node [xshift=1cm,yshift=1cm] at (current page.south west)
        [text width=7cm,fill=red!20,rounded corners,above right]
  {
    This is an absolutely positioned text in the
    lower left corner. No shipout-hackery is used.
  };
\end{tikzpicture}
```

The next example adds a circle in the middle of the page.

```
\begin{tikzpicture}[remember picture,overlay]
  \draw [line width=1mm,opacity=.25]
    (current page.center) circle [radius=3cm];
\end{tikzpicture}
```

The final example overlays some text over the page (depending on where this example is found on the page, the text may also be behind the page).

```
\begin{tikzpicture}[remember picture,overlay]
  \node [rotate=60,scale=10,text opacity=0.2]
    at (current page.center) {Example};
\end{tikzpicture}
```

17.14 Late Code and Late Options

All options given to a node only locally affect this one node. While this is a blessing in most cases, you may sometimes want to cause options to have effects “later” on. The other way round, you may sometimes note

This is an absolutely positioned text in the lower left corner. No shipout-hackery is used.

Example

"only later" that some options should be added to the options of a node. For this, the following version of the `node` path command can be used:

`\path` … `node also[`⟨*late options*⟩`](`⟨*name*⟩`)` … `;`

Note that the ⟨*name*⟩ is compulsory and that *no* text may be given. Also, the ordering of options and node label must be as above.

The effect of the above is the following effect: The node ⟨*name*⟩ must already be existing. Now, the ⟨*late options*⟩ are executed in a local scope. Most of these options will have no effect since you *cannot change the appearance of the node,* that is, you cannot change a red node into a green node using these "late" options. However, giving the `append after command` and `prefix after command` options inside the ⟨*late options*⟩ (directly or indirectly) does have the desired effect: The given path gets executed with the `\tikzlastnode` set to the determined node.

The net effect of all this is that you can provide, say, the `label` option inside the ⟨*options*⟩ to a add a label to a node that has already been constructed.




```
\begin{tikzpicture}
  \node      [draw,circle]       (a) {Hello};
  \node also [label=above:world] (a);
\end{tikzpicture}
```

As explained in Section 14, you can use the options `append after command` and `prefix after command` to add a path after a node. The following macro may be useful there:

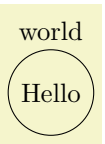
`\tikzlastnode`

Expands to the last node on the path.

Instead of the `node also` syntax, you can also use the following option:

`/tikz/late options=`⟨*options*⟩ (no default)

This option can be given on a path (but not as an argument to a `node` path command) and has the same effect as the `node also` path command. Inside the ⟨*options*⟩, you should use the `name` option to specify the node for which you wish to add late options:




```
\begin{tikzpicture}
  \node      [draw,circle]       (a) {Hello};
  \path [late options={name=a, label=above:world}];
\end{tikzpicture}
```

18 Pics: Small Pictures on Paths

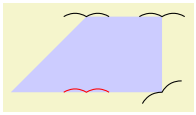
18.1 Overview

A "pic" is a "short picture" (hence the short name…) that can be inserted anywhere in Ti*k*Z picture where you could also insert a node. Similarly to nodes, pics have a "shape" (called *type* to avoid confusion) that someone has defined. Each time a pic of a specified type is used, the type's code is executed, resulting in some drawings to be added to the current picture. The syntax for adding nodes and adding pics to a picture are also very similar. The core difference is that pics are typically more complex than nodes and may consist of a whole bunch of nodes themselves together with complex paths joining them.

As a very simple example, suppose we want to define a pic type `seagull` that just draw "two bumps". The code for this definition is quite easy:

```
\tikzset{
  seagull/.pic={
    % Code for a "seagull". Do you see it?...
    \draw (-3mm,0) to [bend left] (0,0) to [bend left] (3mm,0);
  }
}
```

The first line just tells TEX that you set some Ti*k*Z options for the current scope (which is the whole document); you could put `seagull/.pic=...` anywhere else where Ti*k*Z options are allowed (which is just about anywhere). We have now defined a `seagull` pic type and can use it as follows:



```
\tikz \fill [fill=blue!20]
      (1,1)
   -- (2,2) pic             {seagull}
   -- (3,2) pic             {seagull}
   -- (3,1) pic [rotate=30] {seagull}
   -- (2,1) pic [red]       {seagull};
```

As can be see, defining new types of pics is much easier than defining new shapes for nodes; but see Section 18.3 for the fine details.

Since defining new pics types is easier than defining new node shapes and since using pics is as easy as using nodes, why should you use nodes at all? There are chiefly two reasons:

1. Unlike nodes, pics cannot be referenced later on. You *can* reference nodes that are inside a pic, but not "the pic itself". In particular, you cannot draw lines between pics the way you can draw them between nodes. In general, whenever it makes sense that some drawing could conceivably be connected to other node-like-things, then a node is better than a pic.

2. If pics are used to emulate the full power of a node (which is possible, in principle), they will be slower to construct and take up more memory than a node achieving the same effect.

Despite these drawbacks, pics are an excellent choice for creating highly configurable reusable pieces of drawings that can be inserted into larger contexts.

18.2 The Pic Syntax

`\pic`

Inside `{tikzpicture}` this is an abbreviation for `\path pic`.

The syntax for adding a pic to a picture is very similar to the syntax used for nodes (indeed, internally the same parser code is used). The main difference is that instead of a node contents you provide the picture's type between the braces:

`\path` … `pic` ⟨*foreach statements*⟩ `[`⟨*options*⟩`]` `(`⟨*prefix*⟩`)` `at(`⟨*coordinate*⟩`)` `:`⟨*animation attribute*⟩`={`⟨*options*⟩`}` `{`⟨*pic type*⟩`}` … `;`

Adds a pic to the current Ti*k*Z picture of the specified ⟨*pic type*⟩. The effect is, basically, that some code associated with the ⟨*pic type*⟩ is executed (how this works, exactly, is explained later). This code can consist of arbitrary Ti*k*Z code. As for nodes, the current path will not be modified by this path command, all drawings produced by the code are "external" to the path the same way neither a node nor its border are part of the path on which they are specified.

Just like the `node` command, this path operation is somewhat complex and we go over it step by step.

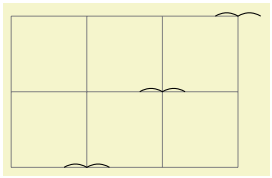
Order of the parts of the specification. Just like for nodes, everything between "pic" and the opening brace of the ⟨*pic type*⟩ is optional and can be given in any order. If there are ⟨*foreach statements*⟩, they must come first, directly following "pic". As for nodes, the "end" of the pic specification is normally detected by the presence of the opening brace. You can, however, use the `pic type` option to specify the pic type as an option.

`/tikz/pic type`=⟨*pic type*⟩ (no default)

This key sets the pic type of the current `pic`. When this option is used inside an option block of a `pic`, the parsing of the `pic` ends immediately and no pic type in braces is expected. (In other words, this option behaves exactly like the `node contents` option and, indeed, the two are interchangeable.)


```
\tikz {
  \path (0,0) pic [pic type = seagull]
        (1,0) pic                      {seagull};
}
```

The location of a pic. Just like nodes, pics are placed at the last position mentioned on the path or, when `at` is used, at a specified position. "Placing" a pic somewhere actually means that the coordinate system is translated (shifted) to this last position. This means that inside of the pic type's code any mentioning of the origin refers to the last position used on the path or to the specified `at`.



```
\tikz { % different ways of placing pics
  \draw [help lines] (0,0) grid (3,2);
  \pic  at (1,0)     {seagull};
  \path (2,1)    pic {seagull};
  \pic  [at={(3,2)}] {seagull};
}
```

As for nodes, except for the described shifting, the coordinate system of a pic is reset prior to executing the pic type's code. This can be changed using the `transform shape` option, which has the same effect as for nodes: The "outer" transformation gets applied to the node:

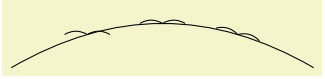

```
\tikz [scale=2] {
  \pic at (0,0)                    {seagull};
  \pic at (1,0) [transform shape] {seagull};
}
```

When the ⟨*options*⟩ contain transformation commands like `scale` or `rotate`, these transformations always apply to the pic:



```
\tikz [rotate=30] {
  \pic at (0,0)             {seagull};
  \pic at (1,0) [rotate=90] {seagull};
}
```

Just like nodes, pics can also be positioned implicitly and, somewhat unsurprisingly, the same rules concerning positioning and sloping apply:



```
\tikz \draw
  (0,0) to [bend left]
           pic [near start]       {seagull}
           pic                    {seagull}
           pic [sloped, near end] {seagull} (4,0);
```

The options of a node. As always, any given ⟨*options*⟩ apply only to the pic and have no effect outside. As for nodes, most "outside" options also apply to the pics, but not the "action" options like `draw` or `fill`. These must be given in the ⟨*options*⟩ of the pic.

The code of a pic. As stated earlier, the main job of a pic is to execute some code in a scope that is shifted according to the last point on the path or to the `at` position specified in the pic. It was also claimed that this code is specified by the ⟨*pic type*⟩. However, this specification is somewhat indirect.

What really happens is the following: When a `pic` is encountered, the current path is suspended and a new internal scope is started. The ⟨*options*⟩ are executed and also the ⟨*pic type*⟩ (as explained in a moment). After all this is done, the code stored in the following key gets executed:

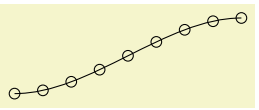
`/tikz/pics/code=`⟨*code*⟩ (no default)

This key stores the ⟨*code*⟩ that should be drawn in the current pic. Normally, setting this key is done by the ⟨*pic type*⟩, but you can also set it in the ⟨*options*⟩ and leave the ⟨*pic type*⟩ empty:


```
\tikz \pic [pics/code={\draw (-3mm,0) to[bend left] (0,0)
                                    to[bend left] (3mm,0);}]
      {}; % no pic type specified
```

Now, how does the ⟨*pic type*⟩ set `pics/code`? It turns out that the ⟨*pic type*⟩ is actually just a list of keys that are executed with the prefix `/tikz/pics/`. In the above examples, this "list of keys" just consisted of the single key "`seagull`" that did not take any arguments, but, in principle, you could provide any arbitrary text understood by `\pgfkeys` here. This means that when we write `pic{seagull}`, Ti*k*Z will execute the key `/tikz/pics/seagull`. It turns out, see Section 18.3, that this key is just a style set to `code={\draw(-3mm,0)...;}`. Thus, `pic{seagull}` will cause the `pics/code` key to be set to the text needed to draw the seagull.

Indeed, you can also use the ⟨*pic type*⟩ simply to set the `code` of the pic. This is useful for cases when you have some code that you "just want to execute, but do not want to define a new pic type". Here is a typical example where we use pics to add some markings to a path:



```
\tikz \draw (0,0) .. controls(1,0) and (2,1) .. (3,1)
  foreach \t in {0, 0.125, ..., 1} {
    pic [pos=\t] {code={\draw circle [radius=2pt];}}
  };
```

In our seagull example, we always explicitly used `\draw` to draw the seagull. This implies that when a user writes something `pic[fill]{seagull}` in the hope of having a "filled" seagull, nothing special will happen: The `\draw` inside the pic explicitly states that the path should be drawn, not filled, and the fact that in the surrounding scope the `fill` option is set has no effect. The following key can be used to change this:

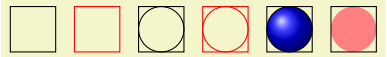
`/tikz/pic actions` (no value)

This key is a style that can be used (only) inside the code of a pic. There, it will set the "action" keys set inside the ⟨*options*⟩ of the pic ("actions" are drawing, filling, shading, and clipping or any combination thereof).

To see how this key works, let us define the following pic:

```
\tikzset{
  my pic/.pic = {
    \path [pic actions] (0,0) circle[radius=3mm];
    \draw (-3mm,-3mm) rectangle (3mm,3mm);
  }
}
```

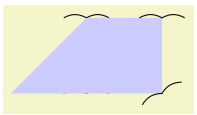
In the code, whether or not the circle gets drawn/filled/shaded depends on which options where given to the `pic` command when it is used. In contrast, the rectangle will always (just) be drawn.



```
\tikz \pic                       {my pic}; \space
\tikz \pic [red]                 {my pic}; \space
\tikz \pic [draw]                {my pic}; \space
\tikz \pic [draw=red]            {my pic}; \space
\tikz \pic [draw, shading=ball] {my pic}; \space
\tikz \pic [fill=red!50]         {my pic};
```

Code executed behind or in front of the path. As for nodes, a pic can be "behind" the current path or "in front of it" and, just as for nodes, the two options `behind path` and `in front of path` are used to specify which is meant. In detail, if `node` and `pic` are both used repeatedly on a path, in

the resulting picture we first see all nodes and pics with the `behind path` option set in the order they appear on the path (nodes and pics are interchangeable in this regard), then comes the path, and then come all nodes and pics that are in front of the path in the order they appeared.



```
\tikz \fill [fill=blue!20]
     (1,1)
  -- (2,2) pic [behind path]      {seagull}
  -- (3,2) pic                    {seagull}
  -- (3,1) pic [rotate=30]        {seagull}
  -- (2,1) pic [red, behind path] {seagull};
```

In contrast to nodes, a pic need not only be completely behind the path or in front of the path as specified by the user. Instead, a pic type may also specify that a certain part of the drawing should always be behind the path and it may specify that a certain other part should always be before the path. For this, the values of the following keys are relevant:

`/tikz/pics/foreground code=`⟨*code*⟩ (no default)

This key stores ⟨*code*⟩ that will always be drawn in front of the current path, even when `behind path` is used. If `behind path` is not used and `code` is (also) set, the code of `code` is drawn first, following by the foreground ⟨*code*⟩.

`/tikz/pics/background code=`⟨*code*⟩ (no default)

Like `foreground code`, only that the ⟨*code*⟩ is always put behind the path, except when the `behind path` option is applied to the pic, then the background code is drawn in front of the "behind path" code.

The foreach statement for pics. As for nodes, a pic specification may start with `foreach`. The effect and semantics are the same as for nodes.


```
\tikz \pic foreach \x in {1,2,3} at (\x,0) {seagull};
```

Styles for pics. The following styles influence how nodes are rendered:

`/tikz/every pic` (style, initially empty)

This style is installed at the beginning of every pic.


```
\begin{tikzpicture}[every pic/.style={scale=2,transform shape}]
  \pic foreach \x in {1,2,3} at (\x,0) {seagull};
\end{tikzpicture}
```

Name scopes. You can specify a ⟨*name*⟩ for a pic using the key `name=`⟨*name*⟩ or by giving the name in parenthesis inside the pic's specification. The effect of this is, for once, quite different from what happens for nodes: All that happens is that `name prefix` is set to ⟨*name*⟩ at the beginning of the pic.

The `name prefix` key was already introduced in the description of the `node` command: It allows you to set some text that is prefixed to all nodes in a scope. For pics this makes particular sense: All nodes defined by a pic's code can be referenced from outside the pic with the prefix provided.

To see how this works, let us add some nodes to the code of the seagull:

```
\tikzset{
  seagull/.pic={
    % Code for a "seagull". Do you see it?...
    \coordinate (-left wing) at (-3mm,0);
    \coordinate (-head)      at (0,0);
    \coordinate (-right wing) at (3mm,0);

    \draw (-left wing) to [bend left] (0,0) (-head) to [bend left] (-right wing);
  }
}
```

Now, we can use it as follows:

```
\tikz {
  \pic (Emma)              {seagull};
  \pic (Alexandra) at (0,1) {seagull};

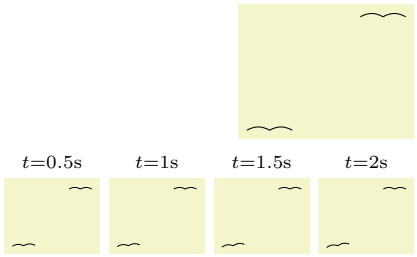
  \draw (Emma-left wing) -- (Alexandra-right wing);
}
```

Sometimes, you may also wish your pic to access nodes outside the pic (typically, because they are given as parameters). In this case, the name prefix gets in the way since the nodes outside the picture do not have this prefix. The trick is to locally reset the name prefix to the value it had outside the picture, which is achieved using the following style:

`/tikz/name prefix ..` (no value)

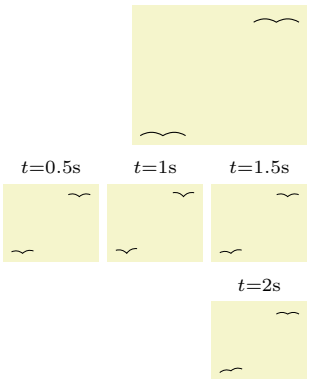
This key is available only inside the code of a pic. There, it (locally) changes the name prefix to the value it had outside the pic. This allows you to access nodes outside the current pic.

Animations for pics. Just as for nodes, you can use the attribute–colon syntax to add an animation to a pic:




```
\usetikzlibrary {animations}
\tikz {
  \pic :rotate={0s="0", 20s="90"} {seagull};
  \pic at (1.5,1.5)               {seagull};
}
```

Naturally, you can also use animations in the code of a picture:




```
\usetikzlibrary {animations}
\begin{tikzpicture} [flapping seagull/.pic={
    \draw (0,0) :path={
      0s= {"{(180:3mm) to [bend left] (0,0) to [bend left] (0:3mm)}"=base},
      1s=  "{(160:3mm) to [bend left] (0,0) to [bend left] (20:3mm)}",
      2s=  "{(180:3mm) to [bend left] (0,0) to [bend left] (0:3mm)}",
      repeats };
  }]
  \pic :rotate={0s="0", 20s="90"} {flapping seagull};
  \pic at (1.5,1.5)               {flapping seagull};
\end{tikzpicture}
```

There are two general purpose keys that pics may find useful:

`/tikz/pic text`=⟨*text*⟩ (no default)

This macro stores the ⟨*text*⟩ in the macro `\tikzpictext`, which is `\let` to `\relax` by default. Setting the `pic text` to some value is the "preferred" way of communicating a (single) piece of text that should become part of a pic (typically of a node). In particular, the `quotes` library maps quoted parameters to this key.

`/tikz/pic text options`=⟨*options*⟩ (no default)

This macro stores the ⟨*options*⟩ in the macro `\tikzpictextoptions`, which is `\let` to the empty string by default. The `quotes` library maps options for quoted parameters to this key.

18.2.1 The Quotes Syntax

When you load the `quotes` library, you can use the "quotes syntax" inside the options of a pic. Recall that for nodes this syntax is used to add a label to a node. For pics, the quotes syntax is used to set the `pic text` key. Whether or not the pic type's code takes this key into consideration is, however, up to the key.

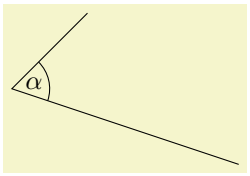
In detail, when the `quotes` library is loaded, each time a key–value pair in a list of options passed to an `pic` starts with `"`, the key–value pair must actually be a string of the following form:

"⟨*text*⟩"'⟨*options*⟩

This string is transformed into the following:

every pic quotes/.try,pic text=⟨*text*⟩, pic text options={⟨*options*⟩}

As example of a pic type that takes these values into account is the `angle` pic type:




```
\usetikzlibrary {angles,quotes}
\tikz \draw (3,0) coordinate (A)
         -- (0,1) coordinate (B)
         -- (1,2) coordinate (C)
            pic [draw, "$\alpha$"] {angle};
```

As described in Section 17.10.4, the apostrophe becomes part of the ⟨*options*⟩, when present. As can be seen above, the following style is executed:

/tikz/every pic quotes (style, initially empty)

18.3 Defining New Pic Types

As explained in the description of the `pic` command, in order to define a new pic type you need to

1. define a key with the path prefix `/tikz/pics` that
2. sets the key `/tikz/pics/code` to the code of the pic.

It turns out that this is easy enough to achieve using styles:

```
\tikzset{
  pics/seagull/.style ={
    % Ok, this is the key that should, when
    % executed, set the code key:
    code = { %
      \draw (...) ... ;
    }
  }
}
```

Even though the above pattern is easy enough, there is a special so-called key handler that allows us to write even simpler code, namely:

```
\tikzset{
  seagull/.pic = {
    \draw (...) ... ;
  }
}
```

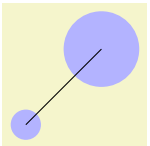
Key handler ⟨*key*⟩`/.pic=`⟨*some code*⟩

This handler can only be used with a key with the prefix `/tikz/`, so you should normally use it only as an option to a Ti*k*Z command or to the `\tikzset` command. It takes the ⟨*key*⟩'s path and, inside that path, it replaces `/tikz/` by `/tikz/pics/` (so, basically, it adds the "missing" `pics` part of the path). Then, it sets up things so that the resulting name to key is a style that executes `code=some code`.

In almost all cases, the `.pic` key handler will suffice to setup keys. However, there are cases where you really need to use the first version using `.style` and `code=`:

- Whenever your pic type needs to set the foreground or the background code.
- In case of complicated arguments given to the keys.

As an example, let us define a simple pic that draws a filled circle behind the path. Furthermore, we make the circle's radius a parameter of the pic:



```
\tikzset{
  pics/my circle/.style = {
    background code = { \fill circle [radius=#1]; }
  }
}
\tikz [fill=blue!30]
  \draw (0,0) pic {my circle=2mm} -- (1,1) pic {my circle=5mm};
```

19 Specifying Graphs

19.1 Overview

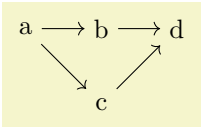
Ti*k*Z offers a powerful path command for specifying how the nodes in a graph are connected by edges and arcs: The `graph` path command, which becomes available when you load the `graphs` library.

Ti*k*Z Library graphs

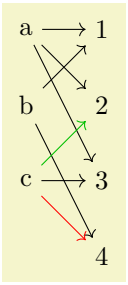
```
\usetikzlibrary{graphs} % LaTeX and plain TeX
\usetikzlibrary[graphs] % ConTeXt
```

The package must be loaded to use the `graph` path command.

In this section, by *graph* we refer to a set of nodes together with some edges (sometimes also called arcs, in case they are directed) such as the following:

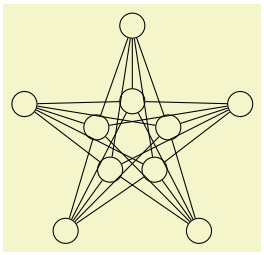



```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b, c} -> d };
```

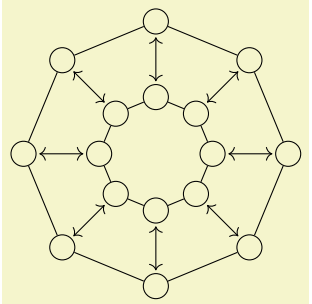



```
\usetikzlibrary {graphs.standard}
\tikz \graph {
  subgraph I_nm [V={a, b, c}, W={1,...,4}];

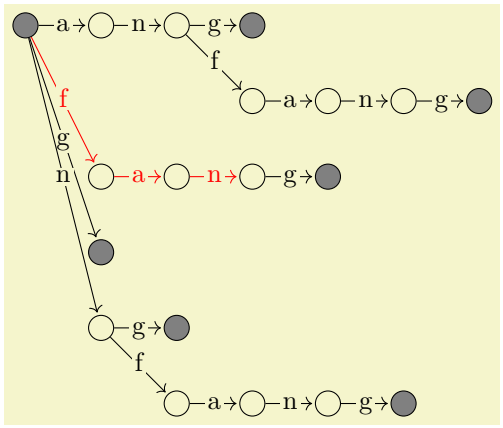
  a -> { 1, 2, 3 };
  b -> { 1, 4 };
  c -> { 2 [>green!75!black], 3, 4 [>red]}
};
```



```
\usetikzlibrary {graphs}
\tikz
  \graph [nodes={draw, circle}, clockwise, radius=.5cm, empty nodes, n=5] {
    subgraph I_n [name=inner] --[complete bipartite]
    subgraph I_n [name=outer]
  };
```

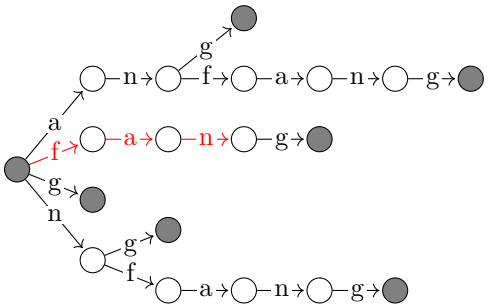


```
\usetikzlibrary {graphs}
\tikz
  \graph [nodes={draw, circle}, clockwise, radius=.75cm, empty nodes, n=8] {
    subgraph C_n [name=inner] <->[shorten <=1pt, shorten >=1pt]
    subgraph C_n [name=outer]
  };
```




```
\usetikzlibrary {graphs}
\tikz [>={To[sep]}, rotate=90, xscale=-1,
       mark/.style={fill=black!50}, mark/.default=]
  \graph [trie, simple,
          nodes={circle,draw},
          edges={nodes={
              inner sep=1pt, anchor=mid,
              fill=graphicbackground}}, % yellowish background
          put node text on incoming edges]
    {
      root[mark] -> {
        a -> n -> {
          g [mark],
          f -> a -> n -> g [mark]
        },
        f -> a -> n -> g [mark],
        g[mark],
        n -> {
          g[mark],
          f -> a -> n -> g[mark]
        }
      },
      { [edges=red] % highlight one path
        root -> f -> a -> n
      }
    };
```

The nodes of a graph are normal Ti*k*Z nodes, the edges are normal lines drawn between nodes. There is nothing in the `graphs` library that you cannot do using the normal `\node` and the `edge` commands. Rather, its purpose is to offer a concise and powerful way of *specifying* which nodes are present and how they are connected. The `graphs` library only offers simple methods for specifying *where* the nodes should be shown, its main strength is in specifying which nodes and edges are present in principle. The problem of finding "good positions on the canvas" for the nodes of a graph is left to *graph drawing algorithms*, which are covered in Part IV of this manual and which are not part of the `graphs` library; indeed, these algorithms can be used also with graphs specified using `node` and `edge` commands. As an example, consider the above drawing of a trie, which is drawn without using the graph drawing libraries. Its layout can be somewhat improved by loading the `layered` graph drawing library, saying `\tikz[layered layout,...`, and then using LuaTEX, resulting in the following drawing of the same graph:



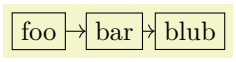
The `graphs` library uses a syntax that is quite different from the normal Ti*k*Z syntax for specifying nodes. The reason for this is that for many medium-sized graphs it can become quite cumbersome to specify all the nodes using `\node` repeatedly and then using a great number of `edge` command; possibly with complicated `\foreach` statements. Instead, the syntax of the `graphs` library is loosely inspired by the DOT format, which is quite useful for specifying medium-sized graphs, with some extensions on top.

19.2 Concepts

The present section aims at giving a quick overview of the main concepts behind the `graph` command. The exact syntax is explained in more detail in later sections.

19.2.1 Concept: Node Chains

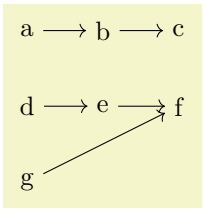
The basic way of specifying a graph is to write down a *node chain* as in the following example:




```
\usetikzlibrary {graphs}
\tikz [every node/.style = draw]
  \graph { foo -> bar -> blub };
```

As can be seen, the text `foo -> bar -> blub` creates three nodes, one with the text `foo`, one with `bar` and one with the text `blub`. These nodes are connected by arrows, which are caused by the `->` between the node texts. Such a sequence of node texts and arrows between them is called a *chain* in the following.

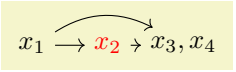
Inside a graph there can be more than one chain:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> b -> c;
  d -> e -> f;
  g -> f;
};
```

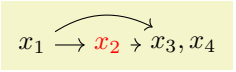
Multiple chains are separated by a semicolon or a comma (both have exactly the same effect). As the example shows, when a node text is seen for the second time, instead of creating a new node, a connection is created to the already existing node.

When a node like `f` is created, both the node name and the node text are identical by default. This is not always desirable and can be changed by using the `as` key or by providing another text after a slash:




```
\usetikzlibrary {graphs}
\tikz \graph {
  x1/$x_1$ -> x2 [as=$x_2$, red] -> x34/{$x_3,x_4$};
  x1 -> [bend left] x34;
};
```

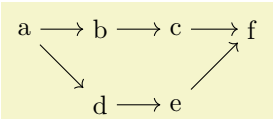
When you wish to use a node name that contains special symbols like commas or dashes, you must surround the node name by quotes. This allows you to use quite arbitrary text as a "node name":




```
\usetikzlibrary {graphs}
\tikz \graph {
  "$x_1$" -> "$x_2$"[red] -> "$x_3,x_4$";
  "$x_1$" ->[bend left] "$x_3,x_4$";
};
```

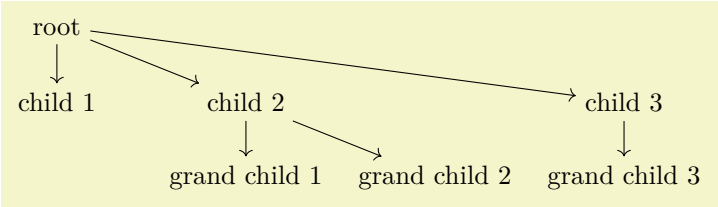
19.2.2 Concept: Chain Groups

Multiple chains that are separated by a semicolon or a comma and that are surrounded by curly braces form what will be called a *chain group* or just a *group*. A group in itself has no special effect. However, things get interesting when you write down a node or even a whole group and connect it to another group. In this case, the "exit points" of the first node or group get connected to the "entry points" of the second node or group:



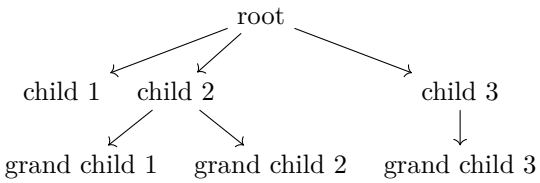

```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> {
    b -> c,
    d -> e
  } -> f
};
```

Chain groups make it easy to create tree structures:



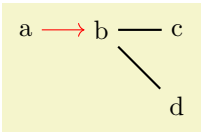

```
\usetikzlibrary {graphs}
\tikz
  \graph [grow down,
          branch right=2.5cm] {
  root -> {
    child 1,
    child 2 -> {
      grand child 1,
      grand child 2
    },
    child 3 -> {
      grand child 3
    }
  }
};
```

As can be seen, the placement is not particularly nice by default, use the algorithms from the graph drawing libraries to get a better layout. For instance, adding `tree layout` to the above code (and `\usetikzlibrary{graphdrawing}` as well as `\usegdlibrary{trees}` to the preamble) results in the following somewhat more pleasing rendering:



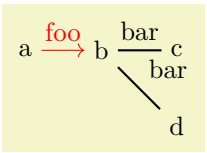
19.2.3 Concept: Edge Labels and Styles

When connectors like `->` or `--` are used to connect nodes or whole chain groups, one or more edges will typically be created. These edges can be styled easily by providing options in square brackets directly after these connectors:



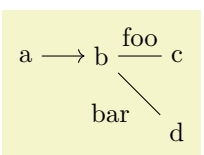

```
\usetikzlibrary {graphs}
\tikz \graph {
  a ->[red] b --[thick] {c, d};
};
```

Using the quotes syntax, see Section 17.10.4, you can even add labels to the edges easily by putting the labels in quotes:



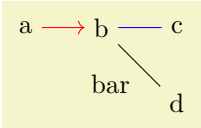

```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a ->[red, "foo"] b --[thick, "bar"] {c, d};
};
```

For the first edge, the effect is as desired, however between `b` and the group `{c,d}` two edges are inserted and the options `thick` and the label option `"bar"` is applied to both of them. While this is the correct and consistent behavior, we typically might wish to specify different labels for the edge going from `b` to `c` and the edge going from `b` to `d`. To achieve this effect, we can no longer specify the label as part of the options of `--`. Rather, we must pass the desired label to the nodes `c` and `d`, but we must somehow also indicate that these options actually "belong" to the edge "leading" to nodes. This is achieved by preceding the options with a greater-than sign:



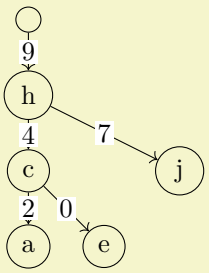

```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a -> b -- {c [> "foo"], d [> "bar"']};
};
```

Symmetrically, preceding the options by `<` causes the options and labels to apply to the "outgoing" edges of the node:




```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a [< red] -> b -- {c [> blue], d [> "bar"']};
};
```

This syntax allows you to easily create trees with special edge labels as in the following example of a treap:

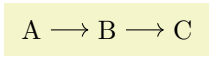



```
\usetikzlibrary {graphs,quotes}
\tikz
  \graph [edge quotes={fill=white,inner sep=1pt},
          grow down, branch right, nodes={circle,draw}] {
    "" -> h [>"9"] -> {
      c [>"4"] -> {
        a [>"2"],
        e [>"0"]
      },
      j [>"7"]
    }
  };
```

19.2.4 Concept: Node Sets

When you write down some node text inside a `graph` command, a new node is created by default unless this node has already been created inside the same `graph` command. In particular, if a node has already been declared outside of the current `graph` command, a new node of the same name gets created.

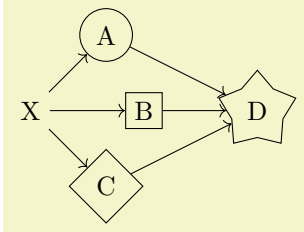
This is not always the desired behavior. Often, you may wish to make nodes part of a graph than have already been defined prior to the use of the `graph` command. For this, simply surround a node name by parentheses. This will cause a reference to be created to an already existing node:




```
\usetikzlibrary {graphs}
\tikz {
  \node (a) at (0,0) {A};
  \node (b) at (1,0) {B};
  \node (c) at (2,0) {C};

  \graph { (a) -> (b) -> (c) };
}
```

You can even go a step further: A whole collection of nodes can all be flagged to belong to a *node set* by adding the option `set=`⟨*node set name*⟩. Then, inside a `graph` command, you can collectively refer to these nodes by surrounding the node set name in parentheses:

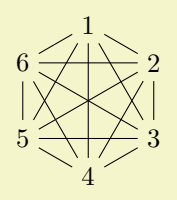



```
\usetikzlibrary {graphs,shapes.geometric}
\tikz [new set=my nodes] {
  \node [set=my nodes, circle,    draw] at (1,1)   {A};
  \node [set=my nodes, rectangle, draw] at (1.5,0) {B};
  \node [set=my nodes, diamond,   draw] at (1,-1)  {C};
  \node (d)            [star,     draw] at (3,0)   {D};

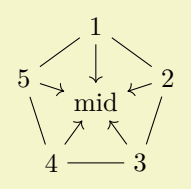
  \graph { X -> (my nodes) -> (d) };
}
```

19.2.5 Concept: Graph Macros

Often, a graph will consist – at least in parts – of standard parts. For instance, a graph might contain a cycle of certain size or a path or a clique. To facilitate specifying such graphs, you can define a *graph macro*. Once a graph macro has been defined, you can use the name of the graph to make a copy of the graph part of the graph currently being specified:




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph K_n [n=6, clockwise] };
```



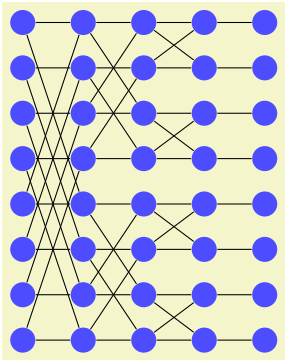

```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph C_n [n=5, clockwise] -> mid };
```

The library `graphs.standard` defines a number of such graphs, including the complete clique K_n on n nodes, the complete bipartite graph $K_{n,m}$ with shores sized n and m, the cycle C_n on n nodes, the path P_n on n nodes, and the independent set I_n on n nodes.

19.2.6 Concept: Graph Expressions and Color Classes

When a graph is being constructed using the `graph` command, it is constructed recursively by uniting smaller graphs to larger graphs. During this recursive union process the nodes of the graph get implicitly *colored* (conceptually) and you can also explicitly assign colors to individual nodes and even change the colors as the graph is being specified. All nodes having the same color form what is called a *color class*.

The power of color class is that special *connector operators* allow you to add edges between nodes having certain colors. For instance, saying `clique=red` at the beginning of a group will cause all nodes that have been flagged as being (conceptually) "red" to be connected as a clique. Similarly, saying `complete bipartite={red}{green}` will cause edges to be added between all red and all green nodes. More advanced connectors, like the `butterfly` connector, allow you to add edges between color classes in a fancy manner.



```
\usetikzlibrary {graphs}
\tikz [x=8mm, y=6mm, circle]
  \graph [nodes={fill=blue!70}, empty nodes, n=8] {
    subgraph I_n [name=A] --[butterfly={level=4}]
    subgraph I_n [name=B] --[butterfly={level=2}]
    subgraph I_n [name=C] --[butterfly]
    subgraph I_n [name=D] --
    subgraph I_n [name=E]
  };
```

19.3 Syntax of the Graph Path Command

19.3.1 The Graph Command

In order to construct a graph, you should use the `graph` path command, which can be used anywhere on a path at any place where you could also use a command like, say, `plot` or `--`.

`\graph`

Inside a `{tikzpicture}` this is an abbreviation for `\path graph`.

`\path` … `graph[`⟨*options*⟩`]`⟨*group specification*⟩ … ;

When this command is encountered on a path, the construction of the current path is suspended (similarly to an `edge` command or a `node` command). In a local scope, the ⟨*options*⟩ are first executed with the key path `/tikz/graphs` using the following command:

`\tikzgraphsset{`⟨*options*⟩`}`

Executes the ⟨*options*⟩ with the path prefix `/tikz/graphs`.

Apart from the keys explained in the following, further permissible keys will be listed during the course of the rest of this section.

`/tikz/graphs/every graph` (style, no value)

This style is executed at the beginning of every `graph` path command prior to the ⟨*options*⟩.

Once the scope has been set up and once the ⟨*options*⟩ have been executed, a parser starts to parse the ⟨*group specification*⟩. The exact syntax of such a group specification in explained in detail in Section 19.3.2. Basically, a group specification is a list of chain specifications, separated by commas or semicolons.

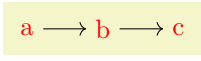
Depending on the content of the ⟨*group specification*⟩, two things will happen:

1. A number of new nodes may be created. These will be inserted into the picture in the same order as if they had been created using multiple `node` path commands at the place where the `graph` path command was used. In other words, all nodes created in a `graph` path command will be painted on top of any nodes created earlier in the path and behind any nodes created later in the path. Like normal nodes, the newly created nodes always lie on top of the path that is currently being created (which is often empty, for instance when the `\graph` command is used).
2. Edges between the nodes may be added. They are added in the same order as if the `edge` command had been used at the position where the `graph` command is being used.

Let us now have a look at some common keys that may be used inside the ⟨*options*⟩:

`/tikz/graphs/nodes=`⟨*options*⟩ (no default)

This option causes the ⟨*options*⟩ to be applied to each newly created node inside the ⟨*group specification*⟩.

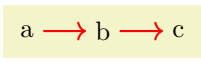



```
\usetikzlibrary {graphs}
\tikz \graph [nodes=red] { a -> b -> c };
```

Multiple uses of this key accumulate.

`/tikz/graphs/edges=`⟨*options*⟩ (no default)

This option causes the ⟨*options*⟩ to be applied to each newly created edge inside the ⟨*group specification*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [edges={red,thick}] { a -> b -> c };
```

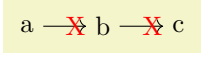
Again, multiple uses of this key accumulate.

`/tikz/graphs/edge=`⟨*options*⟩ (no default)

This is an alias for `edges`.

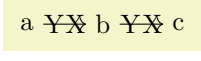
`/tikz/graphs/edge node=`⟨*node specification*⟩ (no default)

This key specifies that the ⟨*node specification*⟩ should be added to each newly created edge as an implicitly placed node.




```
\usetikzlibrary {graphs}
\tikz \graph [edge node={node [red, near end] {X}}] { a -> b -> c };
```

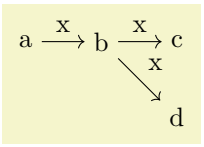
Again, multiple uses of this key accumulate.




```
\usetikzlibrary {graphs}
\tikz \graph [edge node={node [near end] {X}},
              edge node={node [near start] {Y}}] { a -> b -> c };
```

`/tikz/graphs/edge label=`⟨*text*⟩ (no default)

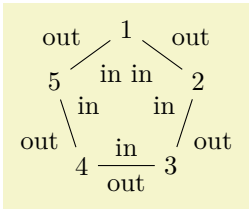
This key is an abbreviation for `edge node=node[auto]{`⟨*text*⟩`}`. The net effect is that the `text` is placed next to the newly created edges.




```
\usetikzlibrary {graphs}
\tikz \graph [edge label=x] { a -> b -> {c,d} };
```

`/tikz/graphs/edge label'=`⟨*text*⟩ (no default)

This key is an abbreviation for `edge node=node[auto,swap]{`⟨*text*⟩`}`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [edge label=out, edge label'=in]
  { subgraph C_n [clockwise, n=5] };
```

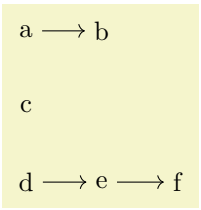
19.3.2 Syntax of Group Specifications

A ⟨*group specification*⟩ inside a `graph` path command has the following syntax:

`{[`⟨*options*⟩`]`⟨*list of chain specifications*⟩`}`

The ⟨*chain specifications*⟩ must contain chain specifications, whose syntax is detailed in the next section, separated by either commas or semicolons; you can freely mix them. It is permissible to use empty lines (which are mapped to `\par` commands internally) to structure the chains visually, they are simply ignored by the parser.

In the following example, the group specification consists of three chain specifications, namely of `a -> b`, then `c` alone, and finally `d -> e -> f`:

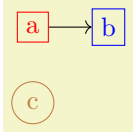



```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> b,
  c;

  d -> e -> f
};
```

The above has the same effect as the more compact group specification `{a->b,c,d->e->f}`.

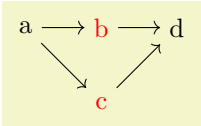
Commas are used to detect where chain specifications end. However, you will often wish to use a comma also inside the options of a single node like in the following example:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a [red, draw] -> b [blue, draw],
  c [brown, draw, circle]
};
```

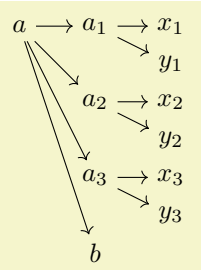
Note that the above example works as expected: The first comma inside the option list of `a` is *not* interpreted as the end of the chain specification "`a [red`". Rather, commas inside square brackets are "protected" against being interpreted as separators of group specifications.

The ⟨*options*⟩ that can be given at the beginning of a group specification are local to the group. They are executed with the path prefix `/tikz/graphs`. Note that for the outermost group specification of a graph it makes no difference whether the options are passed to the `graph` command or whether they are given at the beginning of this group. However, for groups nested inside other groups, it does make a difference:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> { [nodes=red] % the option is local to these nodes:
    b, c
  } ->
  d
};
```

Using foreach. There is special support for the `\foreach` statement inside groups: You may use the statement inside a group specification at any place where a ⟨*chain specification*⟩ would normally go. In this case, the `\foreach` statement is executed and for each iteration the content of the statement's body is treated and parsed as a new chain specification.

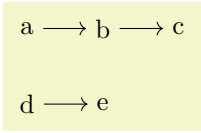



```
\usetikzlibrary {graphs}
\tikz \graph [math nodes, branch down=5mm] {
  a -> {
    \foreach \i in {1,2,3} {
      a_\i -> { x_\i, y_\i }
    },
    b
  }
};
```

Using macros. In some cases you may wish to use macros and TEX code to compute which nodes and edges are present in a group. You cannot use macros in the normal way inside a graph specification since the parser does not expand macros as it scans for the start and end of groups and node names. Rather, only after commas, semicolons, and hyphens have already been detected and only after all other parsing decisions have been made will macros be expanded. At this point, when a macro expands to, say `a,b`, this will not result in two nodes to be created since the parsing is already done. For these reasons, a special key is needed to make it possible to "compute" which nodes should be present in a group.

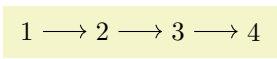
`/tikz/graph/parse=`⟨*text*⟩ (no default)

This key can only be used inside the ⟨*options*⟩ of a ⟨*group specification*⟩. Its effect is that the ⟨*text*⟩ is inserted at the beginning of the current group as if you had entered it there. Naturally, it makes little sense to just write down some static ⟨*text*⟩ since you could just as well directly place it at the beginning of the group. The real power of this command stems from the fact that the keys mechanism allows you to say, for instance, `parse/.expand once` to insert the text stored in some macro into the group.




```
\usetikzlibrary {graphs}
\def\mychain{ a -> b -> c; }
\tikz \graph { [parse/.expand once=\mychain] d -> e };
```

In the following, more fancy example we use a loop to create a chain of dynamic length.




```
\usetikzlibrary {graphs}
\def\mychain#1{
  \def\mytext{1}
  \foreach \i in {2,...,#1} {
    \xdef\mytext{\mytext -> \i}
  }
}
\tikzgraphsset{my chain/.style={
    /utils/exec=\mychain{#1},
    parse/.expand once=\mytext}
}
\tikz \graph { [my chain=4] };
```

Multiple uses of this key accumulate, that is, all the texts given in the different uses is inserted in the order it is given.

19.3.3 Syntax of Chain Specifications

A ⟨*chain specification*⟩ has the following syntax: It consists of a sequence of ⟨*node specifications*⟩, where subsequent node specifications are separated by ⟨*edge specifications*⟩. Node specifications, which typically consist of some text, are discussed in the next section in more detail. They normally represent a single node that is either newly created or exists already, but they may also specify a whole set of nodes.

An ⟨*edge specification*⟩ specifies *which* of the node(s) to the left of the edge specification should be connected to which node(s) to the right of it and it also specifies in which direction the connections go. In the following, we only discuss how the direction is chosen, the powerful mechanism behind choosing which nodes should be connect is detailed in Section 19.7.

The syntax of an edge specification is always one of the following five possibilities:

```
-> [⟨options⟩]
-- [⟨options⟩]
<- [⟨options⟩]
<-> [⟨options⟩]
-!- [⟨options⟩]
```

The first four correspond to a directed edge, an undirected edge, a "backward" directed edge, and a bidirected edge, respectively. The fifth edge specification means that there should be no edge (this specification can be used together with the `simple` option to remove edges that have previously been added, see Section 19.5).

Suppose the nodes ⟨*left nodes*⟩ are to the left of the ⟨*edge specification*⟩ and ⟨*right nodes*⟩ are to the right and suppose we have written `->` between them. Then the following happens:

1. The ⟨*options*⟩ are executed (inside a local scope) with the path `/tikz/graphs`. These options may setup the connector algorithm (see below) and may also use keys like `edge` or `edge label` to specify how the edge should look like. As a convenience, whenever an unknown key is encountered for the path `/tikz/graphs`, the key is passed to the `edge` key. This means that you can directly use options like `thick` or `red` inside the ⟨*options*⟩ and they will apply to the edge as expected.

2. The chosen connector algorithm, see Section 19.7, is used to compute from which of the ⟨*left nodes*⟩ an edge should lead to which of the ⟨*right nodes*⟩. Suppose that (l_1, r_1), …, (l_n, r_n) is the list of node pairs that result (so there should be an edge between l_1 and r_1 and another edge between l_2 and r_2 and so on).

3. For each pair (l_i, r_i) an edge is created. This is done by calling the following key (for the edge specification `->`, other keys are executed for the other kinds of specifications):

   `/tikz/graphs/new ->={`⟨*left node*⟩`}{`⟨*right node*⟩`}{`⟨*edge options*⟩`}{`⟨*edge nodes*⟩`}` (no default)

   This key will be called for a `->` edge specification with the following four parameters:

   (a) ⟨*left node*⟩ is the name of the "left" node, that is, the name of l_i.
   (b) ⟨*right node*⟩ is the name of the right node.
   (c) ⟨*edge options*⟩ are the accumulated options from all calls of `/tikz/graph/edges` in groups that surround the edge specification.
   (d) ⟨*edge nodes*⟩ is text like `node {A} node {B}` that specifies some nodes that should be put as labels on the edge using Ti*k*Z's implicit positioning mechanism.

   By default, the key executes the following code:

   ```
   \path [->,every new ->]
     (⟨left node⟩\tikzgraphleftanchor) edge [⟨edge options⟩] ⟨edge nodes⟩
     (⟨right node⟩\tikzgraphrightanchor);
   ```

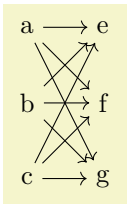
   You are welcome to change the code underlying the key.

   `/tikz/every new ->` (style, no value)

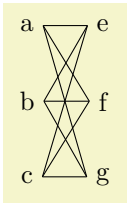
   This key gets executed by default for a `new ->`.

`/tikz/graphs/left anchor=`⟨*anchor*⟩ (no default)

This anchor is used for the node that is to the left of an edge specification. Setting this anchor to the empty string means that no special anchor is used (which is the default). The ⟨*anchor*⟩ is stored in the macro `\tikzgraphleftanchor` with a leading dot.




```
\usetikzlibrary {graphs}
\tikz \graph {
  {a,b,c} -> [complete bipartite] {e,f,g}
};
```




```
\usetikzlibrary {graphs}
\tikz \graph [left anchor=east, right anchor=west] {
  {a,b,c} -- [complete bipartite] {e,f,g}
};
```

/tikz/graphs/right anchor=⟨*anchor*⟩ (no default)

Works like `left anchor`, only for `\tikzgraphrightanchor`.

For the other three kinds of edge specifications, the following keys will be called:

/tikz/graphs/new --={⟨*left node*⟩}{⟨*right node*⟩}{⟨*edge options*⟩}{⟨*edge nodes*⟩} (no default)

This key is called for `--` with the same parameters as above. The only difference in the definition is that in the `\path` command the `->` gets replaced by `-`.

/tikz/every new -- (style, no value)

/tikz/graphs/new <->={⟨*left node*⟩}{⟨*right node*⟩}{⟨*edge options*⟩}{⟨*edge nodes*⟩} (no default)

Called for `<->` with the same parameters as above. The `->` is replaced by `<-`

/tikz/every new <-> (style, no value)

/tikz/graphs/new <-={⟨*left node*⟩}{⟨*right node*⟩}{⟨*edge options*⟩}{⟨*edge nodes*⟩} (no default)

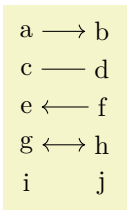
Called for `<-` with the same parameters as above.[4]

/tikz/every new <- (style, no value)

/tikz/graphs/new -!-={⟨*left node*⟩}{⟨*right node*⟩}{⟨*edge options*⟩}{⟨*edge nodes*⟩} (no default)

Called for `-!-` with the same parameters as above. Does nothing by default.

Here is an example that shows the default rendering of the different edge specifications:




```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  a -> b;
  c -- d;
  e <- f;
  g <-> h;
  i -!- j;
};
```

19.3.4 Syntax of Node Specifications

Node specifications are the basic building blocks of a graph specification. There are three different possible kinds of node specifications, each of which has a different syntax:

[4] You might wonder why this key is needed: It seems more logical at first sight to just call `new edge directed` with swapped first parameters. However, a positioning algorithm might wish to take the fact into account that an edge is "backward" rather than "forward" in order to improve the layout. Also, different arrow heads might be used.

Direct Node Specification

"⟨*node name*⟩"/"⟨*text*⟩" [⟨*options*⟩]

(note that the quotation marks are optional and only needed when the ⟨*node name*⟩ contains special symbols)

Reference Node Specification

(⟨*node name or node set name*⟩)

Group Node Specification

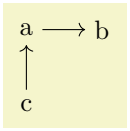
⟨*group specification*⟩

The rule for determining which of the possible kinds is meant is as follows: If the node specification starts with an opening parenthesis, a reference node specification is meant; if it starts with an opening curly brace, a group specification is meant; and in all other cases a direct node specification is meant.

Direct Node Specifications. If after reading the first symbol of a node specification is has been detected to be *direct*, TikZ will collect all text up to the next edge specification and store it as the ⟨*node name*⟩; however, square brackets are used to indicate options and a slash ends the ⟨*node name*⟩ and start a special ⟨*text*⟩ that is used as a rendering text instead of the original ⟨*node name*⟩.

Due to the way the parsing works and due to the restrictions on node names, most special characters are forbidding inside the ⟨*node name*⟩, including commas, semicolons, hyphens, braces, dots, parentheses, slashes, dashes, and more (but spaces, single underscores, and the hat character *are* allowed). To use special characters in the name of a node, you can optionally surround the ⟨*node name*⟩ and/or the ⟨*text*⟩ by quotation marks. In this case, you can use all of the special symbols once more. The details of what happens, exactly, when the ⟨*node name*⟩ is surrounded by quotation marks is explained later; surrounding the ⟨*text*⟩ by quotation marks has essentially the same effect as surrounding it by curly braces.

Once the node name has been determined, it is checked whether the same node name was already used inside the current graph. If this is the case, then we say that the already existing node is *referenced*; otherwise we say that the node is *fresh*.




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> b; % both are fresh
  c -> a; % only c is fresh, a is referenced
};
```

This behavior of deciding whether a node is fresh or referenced can, however, be modified by using the following keys:

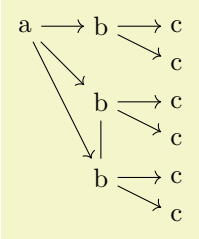
`/tikz/graphs/use existing nodes`=⟨*true or false*⟩ (default `true`)

When this key is set to `true`, all nodes will be considered to the referenced, no node will be fresh. This option is useful if you have already created all the nodes of a graph prior to using the `graph` command and you now only wish to connect the nodes. It also implies that an error is raised if you reference a node which has not been defined previously.

`/tikz/graphs/fresh nodes`=⟨*true or false*⟩ (default `true`)

When this key is set to `true`, all nodes will be considered to be fresh. This option is useful when you create for instance a tree with many identical nodes.

When a node name is encountered that was already used previously, a new name is chosen is follows: An apostrophe (`'`) is appended repeatedly until a node name is found that has not yet been used:

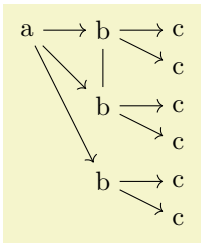



```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  { [fresh nodes]
    a -> {
      b -> {c, c},
      b -> {c, c},
      b -> {c, c},
    }
  },
  b' -- b''
};
```

/tikz/graphs/number nodes=⟨*start number*⟩ (default 1)

When this key is used in a scope, each encountered node name will get appended a new number, starting with ⟨*start*⟩. Typically, this ensures that all node names are different. Between the original node name and the appended number, the setting of the following will be inserted:

/tikz/graphs/number nodes sep=⟨*text*⟩ (no default, initially space)




```
\usetikzlibrary {graphs}
\tikz \graph [branch down=5mm] {
  { [number nodes]
    a -> {
      b -> {c, c},
      b -> {c, c},
      b -> {c, c},
    }
  },
  b 2 -- b 5
};
```

When a fresh node has been detected, a new node is created in the inside a protecting scope. For this, the current placement strategy is asked to compute a default position for the node, see Section 19.9 for details. Then, the command

`\node` (⟨*full node name*⟩) [⟨*node options*⟩] {⟨*text*⟩};

is called. The different parameters are as follows:

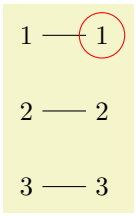
- The ⟨*full node name*⟩ is normally the ⟨*node name*⟩ that has been determined as described before. However, there are two exceptions:

  First, if the ⟨*node name*⟩ is empty (which happens when there is no ⟨*node name*⟩ before the slash), then a fresh internal node name is created and used as ⟨*full node name*⟩. This name is guaranteed to be different from all node names used in this or any other graph. Thus, a direct node starting with a slash represents an anonymous fresh node.

  Second, you can use the following key to prefix the ⟨*node name*⟩ inside the ⟨*full node name*⟩:

  /tikz/graphs/name=⟨*text*⟩ (no default)

  This key prepends the ⟨*text*⟩, followed by a separating symbol (a space by default), to all ⟨*node name*⟩s inside a ⟨*full node name*⟩. Repeated calls of this key accumulate, leading to ever-longer "name paths":



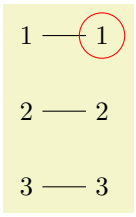

```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph {
    { [name=first]  1, 2, 3} --
    { [name=second] 1, 2, 3}
  };
  \draw [red] (second 1) circle [radius=3mm];
\end{tikzpicture}
```

  Note that, indeed, in the above example six nodes are created even though the first and second set of nodes have the same ⟨*node name*⟩. The reason is that the full names of the six nodes are all different. Also note that only the ⟨*node name*⟩ is used as the node text, not the full name. This can be changed as described later on.

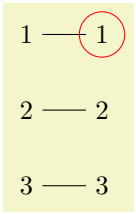
  This key can be used repeatedly, leading to ever longer node names.

  /tikz/graphs/name separator=⟨*symbols*⟩ (no default, initially `\space`)

  Changes the symbol that is used to separate the ⟨*text*⟩ from the ⟨*node name*⟩. The default is `\space`, resulting in a space.




```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph [name separator=] { % no separator
    { [name=first]  1, 2, 3} --
    { [name=second] 1, 2, 3}
  };
  \draw [red] (second1) circle [radius=3mm];
\end{tikzpicture}
```




```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph [name separator=-] {
    { [name=first]  1, 2, 3} --
    { [name=second] 1, 2, 3}
  };
  \draw [red] (second-1) circle [radius=3mm];
\end{tikzpicture}
```

- The ⟨*node options*⟩ are

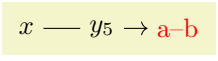
  1. The options that have accumulated in calls to `nodes` from the surrounding scopes.
  2. The local ⟨*options*⟩.

  The options are executed with the path prefix `/tikz/graphs`, but any unknown key is executed with the prefix `/tikz`. This means, in essence, that some esoteric keys are more difficult to use inside the options and that any key with the prefix `/tikz/graphs` will take precedence over a key with the prefix `/tikz`.

- The ⟨*text*⟩ that is passed to the `\node` command is computed as follows: First, you can use the following key to directly set the ⟨*text*⟩:

  `/tikz/graphs/as`=⟨*text*⟩ (no default)

  The ⟨*text*⟩ is used as the text of the node. This allows you to provide a text for the node that differs arbitrarily from the name of the node.


  ```
  \usetikzlibrary {graphs}
  \tikz \graph { a [as=$x$] -- b [as=$y_5$] -> c [red, as={a--b}] };
  ```

  This key always takes precedence over all of the mechanisms described below.

In case the `as` key is not used, a default text is chosen as follows: First, when a direct node specification contains a slash (or, for historical reasons, a double underscore), the text to the right of the slash (or double underscore) is stored in the macro `\tikzgraphnodetext`; if there is no slash, the ⟨*node name*⟩ is stored in `\tikzgraphnodetext`, instead. Then, the current value of the following key is used as ⟨*text*⟩:

`/tikz/graphs/typeset`=⟨*code*⟩ (no default)

The macro or code stored in this key is used as the ⟨*text*⟩ of the node. Inside the ⟨*code*⟩, the following macros are available:

`\tikzgraphnodetext`

This macro expands to the ⟨*text*⟩ to the right of the double underscore or slash in a direct node specification or, if there is no slash, to the ⟨*node name*⟩.

`\tikzgraphnodename`

This macro expands to the name of the current node without the path.

`\tikzgraphnodepath`

This macro expands to the current path of the node. These paths result from the use of the `name` key as described above.

`\tikzgraphnodefullname`

This macro contains the concatenation of the above two.

By default, the typesetter is just set to `\tikzgraphnodetext`, which means that the default text of a node is its name. However, it may be useful to change this: For instance, you might wish that the text of all graph nodes is, say, surrounded by parentheses:

(a) → (b) → (c)

```
\usetikzlibrary {graphs}
\tikz \graph [typeset=(\tikzgraphnodetext)]
  { a -> b -> c };
```

A more advanced macro might take apart the node text and render it differently:

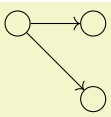
$a_{1,\dots,n}$ ↓ $b_{2,\dots,m}$ ↓ $c_{4,\dots,nm}$

```
\usetikzlibrary {graphs}
\def\mytypesetter{\expandafter\myparser\tikzgraphnodetext\relax}
\def\myparser#1 #2 #3\relax{%
  $#1_{#2,\dots,#3}$
}
\tikz \graph [typeset=\mytypesetter, grow down]
  { a 1 n -> b 2 m -> c 4 nm };
```

The following styles install useful predefined typesetting macros:

`/tikz/graphs/empty nodes` (no value)

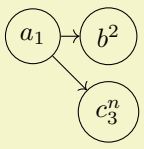
Just sets `typeset` to nothing, which causes all nodes to have an empty text (unless, of course, the `as` option is used):



```
\usetikzlibrary {graphs}
\tikz \graph [empty nodes, nodes={circle, draw}] { a -> {b, c} };
```

`/tikz/graphs/math nodes` (no value)

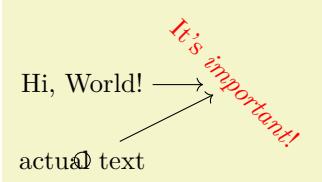
Sets `typeset` to `$\tikzgraphnodetext$`, which causes all nodes names to be typeset in math mode:




```
\usetikzlibrary {graphs}
\tikz \graph [math nodes, nodes={circle, draw}] { a_1 -> {b^2, c_3^n} };
```

If a node is referenced instead of fresh, then this node becomes the node that will be connected by the preceding or following edge specification to other nodes. The ⟨*options*⟩ are executed even for a referenced node, but they cannot be used to change the appearance of the node (because the node exists already). Rather, the ⟨*options*⟩ can only be used to change the logical coloring of the node, see Section 19.7 for details.

Quoted Node Names. When the ⟨*node name*⟩ and/or the ⟨*text*⟩ of a node is surrounded by quotation marks, you can use all sorts of special symbols as part of the text that are normally forbidden:




```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph [grow right=2cm] {
    "Hi, World!"       -> "It's \emph{important}!"[red,rotate=-45];
    "name"/actual text -> "It's \emph{important}!";
  };
  \draw (name) circle [radius=3pt];
\end{tikzpicture}
```

In detail, for the following happens when quotation marks are encountered at the beginning of a node name or its text:

- Everything following the quotation mark up to the next single quotation mark is collected into a macro ⟨*collected*⟩. All sorts of special characters, including commas, square brackets, dashes, and even backslashes are allowed here. Basically, the only restriction is that braces must be balanced.
- A double quotation mark (`""`) does not count as the "next single quotation mark". Rather, it is replaced by a single quotation mark. For instance, `"He said, ""Hello world."""` would be stored inside ⟨*collected*⟩ as `He said, "Hello world."` However, this rule applies only on the outer-most level of braces. Thus, in

  ```
  "He {said, ""Hello world.""}"
  ```

  we would get `He {said, ""Hello world.""}` as ⟨*collected*⟩.
- "The next single quotation mark" refers to the next quotation mark on the current level of braces, so in `"hello {"} world"`, the next quotation mark would be the one following `world`.

Now, once the ⟨*collected*⟩ text has been gather, it is used as follows: When used as ⟨*text*⟩ (what is actually displayed), it is just used "as is". When it is used as ⟨*node name*⟩, however, the following happens: Every "special character" in ⟨*collected*⟩ is replaced by its Unicode name, surrounded by `@`-signs. For instance, if ⟨*collected*⟩ is `Hello, world!`, the ⟨*node name*⟩ is the somewhat longer text `Hello@COMMA@ world@EXCLAMATION MARK@`. Admittedly, referencing such a node from outside the graph is cumbersome, but when you use exactly the same ⟨*collected*⟩ text once more, the same ⟨*node name*⟩ will result. The following characters are considered "special":

```
|$&^~_[](){}/.-,+*'`!":;<=>?@#%\{}
```

These are exactly the Unicode character with a decimal code number between 33 and 126 that are neither digits nor letters.

Reference Node Specifications. A reference node specification is a node specification that starts with an opening parenthesis. In this case, parentheses must surround a ⟨*name*⟩ as in `(foo)`, where `foo` is the ⟨*name*⟩. The following will now happen:

1. It is tested whether ⟨*name*⟩ is the name of a currently active *node set*. This case will be discussed in a moment.
2. Otherwise, the ⟨*name*⟩ is interpreted and treated as a referenced node, but independently of whether the node has already been fresh in the current graph or not. In other words, the node must have been defined either already inside the graph (in which case the parenthesis are more or less superfluous) or it must have been defined outside the current picture.

   The way the referenced node is handled is the same way as for a direct node that is a referenced node.

   If the node does not already exist, an error message is printed.

Let us now have a look at node sets. Inside a `{tikzpicture}` you can locally define a *node set* by using the following key:

`/tikz/new set=`⟨*set name*⟩ (no default)

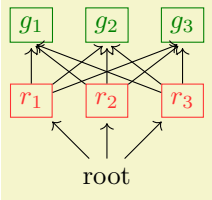
This will setup a node set named ⟨*set name*⟩ within the current scope. Inside the scope, you can add nodes to the node set using the `set` key. If a node set of the same name already exists in the current scope, it will be reset and made empty for the current scope.

Note that this command has the path `/tikz` and is normally used *outside* the `graph` command.

`/tikz/set=`⟨*set name*⟩ (no default)

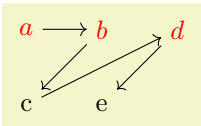
This key can be used as an option with a `node` command. The ⟨*set name*⟩ must be the name of a node set that has previously been created inside some enclosing scope via the `new set` key. The effect is that the current node is added to the node set.

When you use a `graph` command inside a scope where some node set called ⟨*set name*⟩ is defined, then inside this `graph` command you use `(`⟨*set name*⟩`)` to reference *all* of the nodes in the node set. The effect is the same as if instead of the reference to the set name you had created a group specification containing a list of references to all the nodes that are part of the node set.




```
\usetikzlibrary {graphs}
\begin{tikzpicture}[new set=red, new set=green, shorten >=2pt]
  \foreach \i in {1,2,3} {
    \node [draw, red!80,          set=red]   (r\i) at (\i,1) {$r_\i$};
    \node [draw, green!50!black, set=green] (g\i) at (\i,2) {$g_\i$};
  }
  \graph {
    root [xshift=2cm] ->
    (red)             -> [complete bipartite, right anchor=south]
    (green)
  };
\end{tikzpicture}
```

There is an interesting caveat with referencing node sets: Suppose that at the beginning of a graph you just say `(foo);` where `foo` is a set name. Unless you have specified special options, this will cause the following to happen: A group is created whose members are all the nodes of the node set `foo`. These nodes become referenced nodes, but otherwise nothing happens since, by default, the nodes of a group are not connected automatically. However, the referenced nodes have now been referenced inside the graph, you can thus subsequently access them as if they had been defined inside the graph. Here is an example showing how you can create nodes outside a `graph` command and then connect them inside as if they had been declared inside:

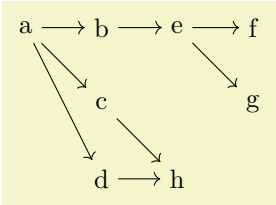



```
\usetikzlibrary {graphs}
\begin{tikzpicture}[new set=import nodes]
  \begin{scope}[nodes={set=import nodes}] % make all nodes part of this set
    \node [red] (a) at (0,1) {$a$};
    \node [red] (b) at (1,1) {$b$};
    \node [red] (d) at (2,1) {$d$};
  \end{scope}

  \graph {
    (import nodes);          % "import" the nodes

    a -> b -> c -> d -> e;  % only c and e are new
  };
\end{tikzpicture}
```

Group Node Specifications. At a place where a node specification should go, you can also instead provide a group specification. Since nodes specifications are part of chain specifications, which in turn are part of group specifications, this is a recursive definition.




```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b,c,d} -> {e -> {f,g}, h} };
```

As can be seen in the above example, when two groups of nodes are connected via an edge specification, it is not immediately obvious which connecting edges are added. This is detailed in Section 19.7.

19.3.5 Specifying Tries

In computer science, a *trie* is a special kind of tree, where for each node and each symbol of an alphabet, there is at most one child of the node labeled with this symbol.

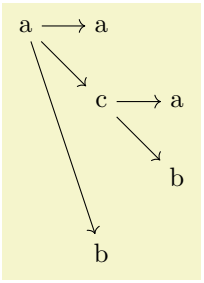
The `trie` key is useful for drawing tries, but it can also be used in other situations. What it does, essentially, is to prepend the node names of all nodes *before* the current node of the current chain to the node's name. This will often make it easier or more natural to specify graphs in which several nodes have the same label.

`/tikz/graphs/trie=`⟨*true or false*⟩ (default `true`, initially `false`)

If this key is set to `true`, after a node has been created on a chain, the `name` key is executed with the node's ⟨*node name*⟩. Thus, all nodes later on this chain have the "path" of nodes leading to this node as their name. This means, in particular, that

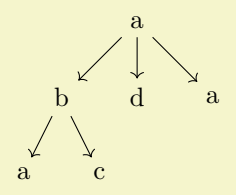
1. two nodes of the same name but in different parts of a chain will be different,
2. while if another chain starts with the same nodes, no new nodes get created.

In total, this is exactly the behavior you would expect of a trie:



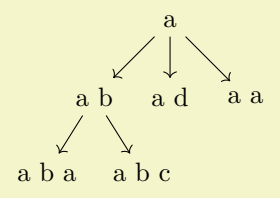

```
\usetikzlibrary {graphs}
\tikz \graph [trie] {
  a -> {
    a,
    c -> {a, b},
    b
  }
};
```

You can even "reiterate" over a path in conjunction with the `simple` option. However, in this case, the default placement strategies will not work and you will need options like `layered layout` from the graph drawing libraries, which need LuaTEX



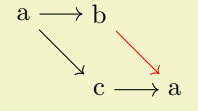

```
\usetikzlibrary {graphs,graphdrawing}\usegdlibrary {layered}
\tikz \graph [trie, simple, layered layout] {
  a -> b -> a,
  a -> b -> c,
  a -> {d,a}
};
```

In the following example, we setup the `typeset` key so that it shows the complete names of the nodes:




```
\usetikzlibrary {graphs,graphdrawing}\usegdlibrary {layered}
\tikz \graph [trie, simple, layered layout,
              typeset=\tikzgraphnodefullname] {
  a -> b -> a,
  a -> b -> c,
  a -> {d,a}
};
```

You can also use the `trie` key locally and later reference nodes using their full name:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { [trie, simple]
    a -> {
      b,
      c -> a
    }
  },
  a b ->[red] a c a
};
```

19.4 Quick Graphs

The graph syntax is powerful, but this power comes at a price: parsing the graph syntax, which is done by TEX, can take some time. Normally, the parsing is fast enough that you will not notice it, but it can be bothersome when you have graphs with hundreds of nodes as happens frequently when nodes are generated algorithmically by some other program. Fortunately, when another program generated a graph specification, we typically do not need the full power of the graph syntax. Rather, a small subset of the graph syntax would suffice that allows to specify nodes and edges. For these reasons, the is a special "quick" version of the graph syntax.

Note, however, that using this syntax will usually at most halve the time needed to parse a graph. Thus, it really mostly makes sense in conjunction with large, algorithmically generated graphs.

/tikz/graphs/quick (no value)

When you provide this key with a graph, the syntax of graph specifications gets restricted. You are no longer allowed to use certain features of the graph syntax; but all features that are still allowed are also allowed in the same way when you do not provide the `quick` option. Thus, leaving out the `quick` option will never hurt.

Since the syntax is so severely restricted, it is easier to explain which aspects of the graph syntax *will* still work:

1. A quick graph consists of a sequence of either nodes, edges sequences, or groups. These are separated by commas or semicolons.
2. Every node is of the form

   `"`⟨*node name*⟩`"/"`⟨*node text*⟩`"[`⟨*options*⟩`]`

   The quotation marks are mandatory. The part `/"`⟨*node text*⟩`"` may be missing, in which case the node name is used as the node text. The ⟨*options*⟩ may also be missing. The ⟨*node name*⟩ may not contain any "funny" characters (unlike in the normal graph command).
3. Every chain is of the form

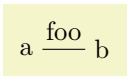
   ⟨*node spec*⟩ ⟨*connector*⟩ ⟨*node spec*⟩ ⟨*connector*⟩ …⟨*connector*⟩ ⟨*node spec*⟩`;`

   Here, the ⟨*node spec*⟩ are node specifications as described above, the ⟨*connector*⟩ is one of the four connectors `->`, `<-`, `--`, and `<->` (the connector `-!-` is not allowed since the `simple` option is also not allowed). Each connector may be followed by options in square brackets. The semicolon may be replaced by a comma.
4. Every group is of the form

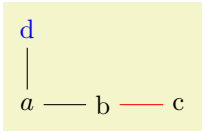
   `{ [`⟨*options*⟩`]` ⟨*chains and groups*⟩ `};`

   The ⟨*options*⟩ are compulsory. The semicolon can, again, be replaced by a comma.
5. The `number nodes` option will work as expected.

Here is a typical way this syntax might be used:




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [quick] { "a" --["foo"] "b"[x=1] };
```




```
\usetikzlibrary {graphs}
\tikz \graph [quick] {
  "a"/"$a$" -- "b"[x=1] --[red] "c"[x=2];
  { [nodes=blue] "a" -- "d"[y=1]; };
};
```

Let us now have a look at the most important things that will *not* work when the `quick` option is used:

- Connecting a node and a group as in `a->{b,c}`.
- Node names without quotation marks as in `a--b`.
- Everything described in subsequent subsections, which includes subgraphs (graph macros), graph sets, graph color classes, anonymous nodes, the `fresh nodes` option, sublayouts, simple graphs, edge annotations.
- Placement strategies – you either have to define all node positions explicitly using `at=` or `x=` and `y=` or you must use a graph drawing algorithm like `layered layout`.

19.5 Simple Versus Multi-Graphs

The `graphs` library allows you to construct both simple graphs and multi-graphs. In a simple graph there can be at most one edge between any two vertices, while in a multi-graph there can be multiple edges (hence the name). The two keys `multi` and `simple` allow you to switch (even locally inside on of the graph's scopes) between which kind of graph is being constructed. By default, the `graph` command produces a multi-graph since these are faster to construct.

/tikz/graphs/multi (no value)

When this edge is set for a whole graph (which is the default) or just for a group (which is useful if the whole graph is simple in general, but a part is a multi-graph), then when you specify an edge between two nodes several times, several such edges get created:



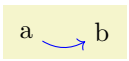

```
\usetikzlibrary {graphs}
\tikz \graph [multi] { % "multi" is not really necessary here
  a ->[bend left,  red]  b;
  a ->[bend right, blue] b;
};
```

In case `multi` is used for a scope inside a larger scope where the `simple` option is specified, then inside the local `multi` scope edges are immediately created and they are completely ignored when it comes to deciding which kind of edges should be present in the surrounding simple graph. From the surrounding scope's point of view it is as if the local `multi` graph contained no edges at all.

This means, in particular, that you can use the `multi` option with a single edge to "enforce" this edge to be present in a simple graph.

/tikz/graphs/simple (no value)

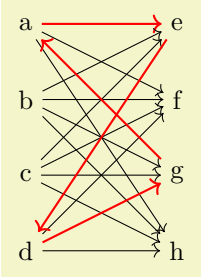
In contrast a multi-graph, in a simple graph, at most one edge gets created for every pair of vertices:




```
\usetikzlibrary {graphs}
\tikz \graph [simple]{
  a ->[bend left,  red]  b;
  a ->[bend right, blue] b;
};
```

As can be seen, the second edge "wins" over the first edge. The general rule is as follows: In a simple graph, whenever an edge between two vertices is specified multiple times, only the very last specification and its options will actually be executed.

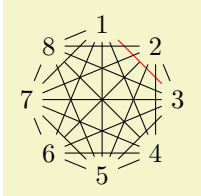
The real power of the `simple` option lies in the fact that you can first create a complicated graph and then later redirect and otherwise modify edges easily:




```
\usetikzlibrary {graphs}
\tikz \graph [simple, grow right=2cm] {
  {a,b,c,d} ->[complete bipartite] {e,f,g,h};

  { [edges={red,thick}] a -> e -> d -> g -> a };
};
```

One particularly interesting kind of edge specification for a simple graph is `-!-`. Recall that this is used to indicate that "no edge" should be added between certain nodes. In a multi-graph, this key usually has no effect (unless the key `new -!-` has been redefined) and is pretty superfluous. In a simple graph, however, it counts as an edge kind and you can thus use it to remove an edge that been added previously:




```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple] {
  subgraph K_n [n=8, clockwise];
  % Get rid of the following edges:
  1 -!- 2;
  3 -!- 4;
  6 -!- 8;
  % And make one edge red:
  1 --[red] 3;
};
```

Creating a graph such as the above in other fashions is pretty awkward.

For every unordered pair $\{u, v\}$ of vertices at most one edge will be created in a simple graph. In particular, when you say `a -> b` and later also `a <- b`, then only the edge `a <- b` will be created. Similarly, when you say `a -> b` and later `b -> a`, then only the edge `b -> a` will be created.

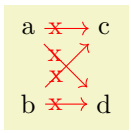
The power of the `simple` command comes at a certain cost: As the graph is being constructed, a (sparse) array is created that keeps track for each edge of the last edge being specified. Then, at the end of the scope containing the `simple` command, for every pair of vertices the edge is created. This is implemented by two nested loops iterating over all possible pairs of vertices – which may take quite a while in a graph of, say, 1000 vertices. Internally, the `simple` command is implemented as an operator that adds the edges when it is called, but this should be unimportant in normal situations.

19.6 Graph Edges: Labeling and Styling

When the `graphs` library creates an edge between two nodes in a graph, the appearance (called "styling" in Ti*k*Z) can be specified in different ways. Sometimes you will simply wish to say "the edges between these two groups of node should be red", but sometimes you may wish to say "this particular edge going into this node should be red". In the following, different ways of specifying such styling requirements are discussed. Note that adding labels to edges is, from Ti*k*Z's point of view, almost the same as styling edges, since they are also specified using options.

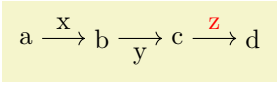
19.6.1 Options For All Edges Between Two Groups

When you write `... ->[options] ...` somewhere inside your graph specification, this typically cause one or more edges to be created between the nodes in the chain group before the `->` and the nodes in the chain group following it. The `options` are applied to all of them. In particular, if you use the `quotes` library and you write some text in quotes inside the `options`, this text will be added as a label to each edge:




```
\usetikzlibrary {graphs,quotes}
\tikz
  \graph [edge quotes=near start] {
    { a, b } -> [red, "x", complete bipartite] { c, d };
  };
```

As documented in the `quotes` library in more detail, you can easily modify the appearance of edge labels created using the quotes syntax by adding options after the closing quotes:

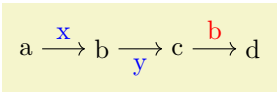



```
\usetikzlibrary {graphs,quotes}
\tikz \graph {
  a ->["x"] b ->["y"'] c ->["z" red] d;
};
```

The following options make it easy to setup the styling of nodes created in this way:

`/tikz/graphs/edge quotes=`⟨*options*⟩ (no default)

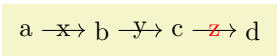
A shorthand for setting the style `every edge quotes` to ⟨*options*⟩.




```
\usetikzlibrary {graphs,quotes}
  \tikz \graph [edge quotes={blue,auto}] {
  a ->["x"] b ->["y"'] c ->["b" red] d;
};
```

`/tikz/graphs/edge quotes center` (no value)

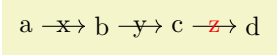
A shorthand for `edge quotes` to `anchor=center`.




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [edge quotes center] {
  a ->["x"] b ->["y"] c ->["z" red] d;
};
```

`/tikz/graphs/edge quotes mid` (no value)

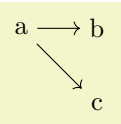
A shorthand for `edge quotes` to `anchor=mid`.




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [edge quotes mid] {
  a ->["x"] b ->["y"] c ->["z" red] d;
};
```

19.6.2 Changing Options For Certain Edges

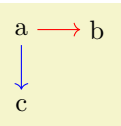
Consider the following tree-like graph:




```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b,c} };
```

Suppose we wish to specify that the edge from `a` to `b` should be red, while the edge from `a` to `c` should be blue. The difficulty lies in the fact that *both* edges are created by the single `->` operator and we can only add one of these option `red` or `blue` to the operator.

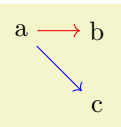
There are several ways to solve this problem. First, we can simply split up the specification and specify the two edges separately:




```
\usetikzlibrary {graphs}
\tikz \graph {
  a -> [red]  b;
  a -> [blue] c;
};
```

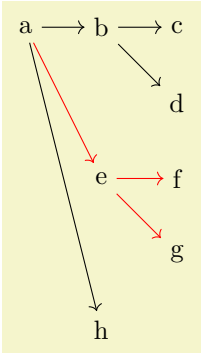
While this works quite well, we can no longer use the nice chain group syntax of the `graphs` library. For the rather simple graph `a->{b,c}` this is not a big problem, but if you specify a tree with, say, 30 nodes it is really worthwhile being able to specify the tree "in its natural form in the TEX code" rather than having to list all of the edges explicitly. Also, as can be seen in the above example, the node placement is changed, which is not always desirable.

One can sidestep this problem using the `simple` option: This option allows you to first specify a graph and then, later on, replace edges by other edges and, thereby, provide new options:




```
\usetikzlibrary {graphs}
\tikz \graph [simple] {
  a -> {b,c};
  a -> [red]  b;
  a -> [blue] c;
};
```

The first line is the original specification of the tree, while the following two lines replace some edges of the tree (in this case, all of them) by edges with special options. While this method is slower and in the above example creates even longer code, it is very useful if you wish to, say, highlight a path in a larger tree: First specify the tree normally and, then, "respecify" the path or paths with some other edge options in force. In the following example, we use this to highlight a whole subtree of a larger tree:




```
\usetikzlibrary {graphs}
\tikz \graph [simple] {
  % The larger tree, no special options in force
  a -> {
    b -> {c,d},
    e -> {f,g},
    h
  },
  { [edges=red] % Now highlight a part of the tree
    a -> e -> {f,g}
  }
};
```

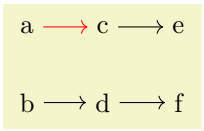
19.6.3 Options For Incoming and Outgoing Edges

When you use the syntax `... ->[options] ...` to specify options, you specify options for the "connections between two sets of nodes". In many cases, however, it will be more natural to specify options "for the edges lead to or coming from a certain node" and you will want to specify these options "at the node". Returning to the example of the graph `a->{b,c}` where we want a red edge between `a` and `b` and a blue edge between `a` and `c`, this could also be phrased as follows: "Make the edge leading to `b` red and make the edge leading to `c` blue".

For this situation, the `graphs` library offers a number of special keys, which are documented in the following. However, most of the time you will not use these keys directly, but, rather, use a special syntax explained in Section 19.6.4.

`/tikz/graphs/target edge style`=⟨*options*⟩ (no default)

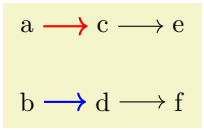
This key can (only) be used with a *node* inside a graph specification. When used, the ⟨*options*⟩ will be added to every edge that is created by a connector like `->` in which the node is a *target*. Consider the following example:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge style=red], d } ->
  { e, f }
};
```

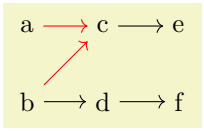
In the example, only when the edge from `a` to `c` is created, `c` is the "target" of the edge. Thus, only this edge becomes red.

When an edge already has options set directly, the ⟨*options*⟩ are executed after these direct options, thus, they "overrule" them:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } -> [blue, thick]
  { c [target edge style=red], d } ->
  { e, f }
};
```

The ⟨*options*⟩ set in this way will stay attached to the node, so also for edges created later on that lead to the node will have these options set:

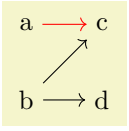



```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge style=red], d } ->
  { e, f },
  b -> c
};
```

Multiple uses of this key accumulate. However, you may sometimes also wish to "clear" these options for a key since at some later point you no longer wish the ⟨*options*⟩ to be added when some further edges are added. This can be achieved using the following key:

`/tikz/graphs/target edge clear` (no value)

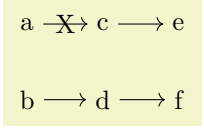
Clears all ⟨*options*⟩ for edges with the node as a target and also edge labels (see below) for this node.




```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge style=red], d },
  b -> c[target edge clear]
};
```

`/tikz/graphs/target edge node`=⟨*node specification*⟩ (no default)

This key works like `target edge style`, only the ⟨*node specification*⟩ will not be added as options to any newly created edges with the current node as their target, but rather it will be added as a node specification.

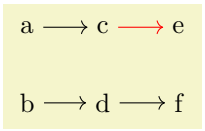



```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [target edge node=node{X}], d } ->
  { e, f }
};
```

As for `target edge style` multiple uses of this key accumulate and the key `target edge clear` will (also) clear all target edge nodes that have been set for a node earlier on.

`/tikz/graphs/source edge style`=⟨*options*⟩ (no default)

Works exactly like `target edge style`, only now the ⟨*options*⟩ are only added when the node is a source of a newly created edge:




```
\usetikzlibrary {graphs}
\tikz \graph {
  { a, b } ->
  { c [source edge style=red], d } ->
  { e, f }
};
```

If both for the source and also for the target of an edge ⟨*options*⟩ have been specified, the options are applied in the following order:

1. First come the options from the edge itself.
2. Then come the options contributed by the source node using this key.
3. Then come the options contributed by the target node using `target node style`.




```
\usetikzlibrary {graphs}
\tikz \graph {
  a [source edge style=red] ->[green]
  b [target edge style=blue]  % blue wins
};
```

`/tikz/graphs/source edge node`=⟨*node specification*⟩ (no default)

Works like `source edge style` and `target edge node`.

`/tikz/graphs/source edge clear`=⟨*node specification*⟩ (no default)

Works like `target edge clear`.

19.6.4 Special Syntax for Options For Incoming and Outgoing Edges

The keys `target node style` and its friends are powerful, but a bit cumbersome to write down. For this reason, the `graphs` library introduces a special syntax that is based on what I call the "first-char syntax" of keys. Inside the options of a node inside a graph, the following special rules apply:

1. Whenever an option starts with `>`, the rest of the options are passed to `target edge style`. For instance, when you write `a[>red]`, then this has the same effect as if you had written

   ```
   a[target edge style={red}]
   ```

2. Whenever an options starts with `<`, the rest of the options are passed to `source edge style`.
3. In both of the above case, in case the options following the `>` or `<` sign start with a quote, the created edge label is passed to `source edge node` or `target edge node`, respectively.

   This is exactly what you want to happen.

Additionally, the following styles provide shorthands for "clearing" the target and source options:

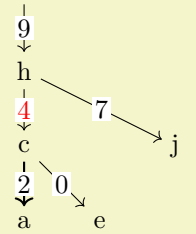
`/tikz/graphs/clear >` (no value)

A more easy-to-remember shorthand for `target edge clear`.

`/tikz/graphs/clear <` (no value)

A more easy-to-remember shorthand for `source edge clear`.

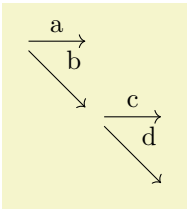
These mechanisms make it especially easy to create trees in which the edges are labeled in some special way:




```
\usetikzlibrary {graphs,quotes}
\tikz
  \graph [edge quotes={fill=white,inner sep=1pt},
          grow down, branch right] {
    / -> h ["9"] -> {
      c ["4" text=red,] -> {
        a ["2", >thick],
        e ["0"]
      },
      j ["7"]
    }
  };
```

19.6.5 Placing Node Texts on Incoming Edges

Normally, the text of a node is shown (only) inside the node. In some case, for instance when drawing certain kind of trees, the nodes themselves should not get any text, but rather the edge leading to the node should be labeled as in the following example:




```
\usetikzlibrary {graphs,quotes}
\tikz \graph [empty nodes]
{
  root -> {
    a ["a"],
    b ["b"] -> {
      c ["c"],
      d ["d"]
    }
  }
};
```

As the example shows, it is a bit cumbersome that we have to label the nodes and then specify the same text once more using the incoming edge syntax.

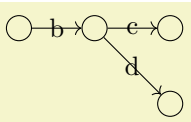
For these cases, it would be better if the text of the node where not used with the node but, rather, be passed directly to the incoming or the outgoing edge. The following styles do exactly this:

`/tikz/graphs/put node text on incoming edges=`⟨*options*⟩ (no default)

When this key is used with a node or a group, the following happens:

1. The command `target edge node={node[`⟨*options*⟩`]{\tikzgraphnodetext}}` is executed. This means that all incoming edges of the node get a label with the text that would usually be displayed in the node. You can use keys like `math nodes` normally.
2. The command `as={}` is executed. This means that the node itself will display nothing.

Here is an example that show how this command is used.




```
\usetikzlibrary {graphs}
\tikz \graph [put node text on incoming edges,
              math nodes, nodes={circle,draw}]
  { a -> b -> {c, d} };
```

`/tikz/graphs/put node text on outgoing edges=`⟨*options*⟩ (no default)

Works like the previous key, only with `target` replaced by `source`.

19.7 Graph Operators, Color Classes, and Graph Expressions

Ti*k*Z's `graph` command employs a powerful mechanism for adding edges between nodes and sets of nodes. To a graph theorist, this mechanism may be known as a *graph expression*: A graph is specified by starting with small graphs and then applying *operators* to them that form larger graphs and that connect and recolor colored subsets of the graph's node in different ways.

19.7.1 Color Classes

TikZ keeps track of a *(multi)coloring* of the graph as it is being constructed. This does not mean that the actual color of the nodes on the page will be different, rather, in the following we refer to "logical" colors in the way graph theoreticians do. These "logical" colors are only important while the graph is being constructed and they are "thrown away" at the end of the construction. The actual ("physical") colors of the nodes are set independently of these logical colors.

As a graph is being constructed, each node can be part of one or more overlapping *color classes*. So, unlike what is sometimes called a *legal coloring*, the logical colorings that TikZ keeps track of may assign multiple colors to the same node and two nodes connected by an edge may well have the same color.

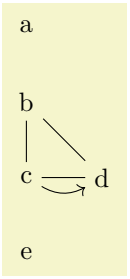
Color classes must be declared prior to use. This is done using the following key:

`/tikz/graphs/color class`=⟨*color class name*⟩ (no default)

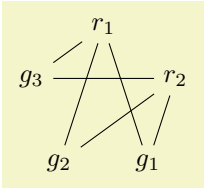
This sets up a new color class called ⟨*color class name*⟩. Nodes and whole groups of nodes can now be colored with ⟨*color class name*⟩. This is done using the following keys, which become available inside the current scope:

`/tikz/graphs/`⟨*color class name*⟩ (no value)

This key internally uses the `operator` command to setup an operator that will cause all nodes of the current group to get the "logical color" ⟨*color class name*⟩. Nodes retain this color in all encompassing scopes, unless it is explicitly changed (see below) or unset (again, see below).



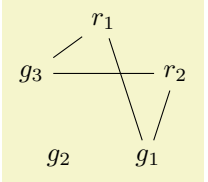

```
\usetikzlibrary {graphs}
\tikz \graph [color class=red] {
  [cycle=red]  % causes all "logically" red nodes to be connected in
               % a cycle
  a,
  b [red],
  { [red] c ->[bend right] d },
  e
};
```




```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green,
              math nodes, clockwise, n=5] {
  [complete bipartite={red}{green}]
  { [red]   r_1, r_2 },
  { [green] g_1, g_2, g_3 }
};
```

`/tikz/graphs/not` ⟨*color class name*⟩ (no value)

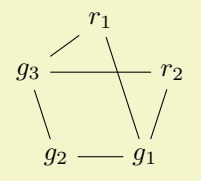
Sets up an operator for the current scope so that all nodes in it loose the color ⟨*color class name*⟩. You can also use `!`⟨*color class name*⟩ as an alias for this key.




```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green,
              math nodes, clockwise, n=5] {
  [complete bipartite={red}{green}]
  { [red]   r_1, r_2 },
  { [green] g_1, g_2, g_3 },
  g_2 [not green]
};
```

`/tikz/graphs/recolor` ⟨*color class name*⟩ `by`=⟨*new color*⟩ (no default)

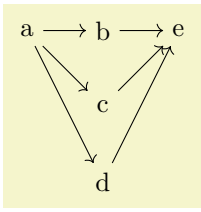
Causes all keys having color ⟨*color class name*⟩ to get ⟨*new color*⟩ instead. They loose having color ⟨*color class name*⟩, but other colors are not affected.



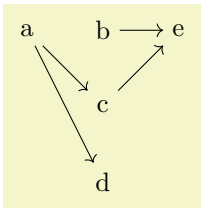

```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green,
              math nodes, clockwise, n=5] {
  [complete bipartite={red}{green}]
  { [red]   r_1, r_2 },
  { [green] g_1, g_2, g_3 },
  g_2 [recolor green by=red]
};
```

The following color classes are available by default:

- Color class `all`. Every node is part of this class by default. This is useful to access all nodes of a (sub)graph, since you can simply access all nodes of this color class.
- Color classes `source` and `target`. These classes are used to identify nodes that lead "into" a group of nodes and nodes from which paths should "leave" the group. Details on how these colors are assigned are explained in Section 19.7.3. By saying `not source` or `not target` with a node, you can influence how it is connected:




```
\usetikzlibrary {graphs}
\tikz \graph { a -> { b, c, d } -> e };
```




```
\usetikzlibrary {graphs}
\tikz \graph { a -> { b[not source], c, d[not target] } -> e };
```

- Color classes `source'` and `target'`. These are temporary colors that are also explained in Section 19.7.3.

19.7.2 Graph Operators on Groups of Nodes

Recall that the `graph` command constructs graphs recursively from nested ⟨*group specifications*⟩. Each such ⟨*group specification*⟩ describes a subset of the nodes of the final graph. A *graph operator* is an algorithm that gets the nodes of a group as input and (typically) adds edges between these nodes in some sensible way. For instance, the `clique` operator will simply add edges between all nodes of the group.

`/tikz/graphs/operator`=⟨*code*⟩ (no default)

This key has an effect in three places:

1. It can be used in the ⟨*options*⟩ of a ⟨*direct node specification*⟩.
2. It can be used in the ⟨*options*⟩ of a ⟨*group specification*⟩.
3. It can be used in the ⟨*options*⟩ of an ⟨*edge specification*⟩.

The first case is a special case of the second, since it is treated like a group specification containing a single node. The last case is more complicated and discussed in the next section. So, let us focus on the second case.

Even though the ⟨*options*⟩ of a group are given at the beginning of the ⟨*group specification*⟩, the ⟨*code*⟩ is only executed when the group has been parsed completely and all its nodes have been identified. If you use the `operator` multiple times in the ⟨*options*⟩, the effect accumulates, that is, all code passed to the different calls of `operator` gets executed in the order it is encountered.

The ⟨*code*⟩ can do "whatever it wants", but it will typically add edges between certain nodes. You can configure what kind of edges (directed, undirected, etc.) are created by using the following keys:

/tikz/graphs/default edge kind=⟨*value*⟩ (no default, initially --)

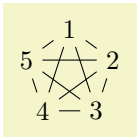
This key stores one of the five edge kinds --, <-, ->, <->, and -!-. When an operator wishes to create a new edge, it should typically set

```
\tikzgraphsset{new \pfkeysvalueof{/tikz/graphs/default edge kind}=...}
```

While this key can be set explicitly, it may be more convenient to use the abbreviating keys listed below. Also, this key is automatically set to the current value of ⟨*edge specification*⟩ when a joining operator is called, see the discussion of joining operators in Section 19.7.3.

/tikz/graphs/-- (no value)

Sets the `default edge kind` to --.




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph K_n [--, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/-> (no value)

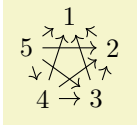
Sets the `default edge kind` to ->.




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph K_n [->, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/<- (no value)

Sets the `default edge kind` to <-.




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph K_n [<-, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/<-> (no value)

Sets the `default edge kind` to <->.




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph K_n [<->, n=5, clockwise, radius=6mm] };
```

/tikz/graphs/-!- (no value)

Sets the `default edge kind` to -!-.

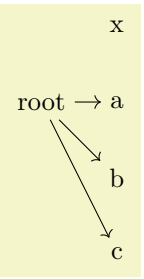
When the ⟨*code*⟩ of an operator is executed, the following commands can be used to find the nodes that should be connected:

\tikzgraphforeachcolorednode{⟨*color name*⟩}{⟨*macro*⟩}

When this command is called inside ⟨*code*⟩, the following will happen: Ti*k*Z will iterate over all nodes inside the just-specified group that have the color ⟨*color name*⟩. The order in which they are iterated over is the order in which they appear inside the group specification (if a node is

encountered several times inside the specification, only the first occurrence counts). Then, for each node the ⟨*macro*⟩ is executed with the node's name as the only argument.

In the following example we use an operator to connect every node colored `all` inside the subgroup to he node `root`.




```
\usetikzlibrary {graphs}
\def\myconnect#1{\tikzset{graphs/new ->={root}{#1}{}{}}}

\begin{tikzpicture}
  \node (root) at (-1,-1) {root};

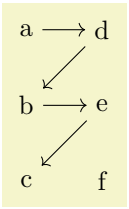
  \graph {
    x,
    {
      [operator=\tikzgraphforeachcolorednode{all}{\myconnect}]
      a, b, c
    }
  };
\end{tikzpicture}
```

`\tikzgraphpreparecolor{`⟨*color name*⟩`}{`⟨*counter*⟩`}{`⟨*prefix*⟩`}`

This command is used to "prepare" the nodes of a certain color for random access. The effect is the following: It is counted how many nodes there are having color ⟨*color name*⟩ in the current group and the result is stored in ⟨*counter*⟩. Next, macros named ⟨*prefix*⟩`1`, ⟨*prefix*⟩`2`, and so on are defined, that store the names of the first, second, third, and so on node having the color ⟨*color name*⟩.

The net effect is that after you have prepared a color, you can quickly iterate over them. This is especially useful when you iterate over several color at the same time.

As an example, let us create an operator then adds a zig-zag path between two color classes:




```
\usetikzlibrary {graphs}
\newcount\leftshorecount   \newcount\rightshorecount
\newcount\mycount          \newcount\myothercount
\def\zigzag{
  \tikzgraphpreparecolor{left shore}\leftshorecount{left shore prefix}
  \tikzgraphpreparecolor{right shore}\rightshorecount{right shore prefix}
  \mycount=0\relax
  \loop
    \advance\mycount by 1\relax%
    % Add the "forward" edge
    \tikzgraphsset{new ->=
      {\csname left shore prefix\the\mycount\endcsname}
      {\csname right shore prefix\the\mycount\endcsname}{}{}}
    \myothercount=\mycount\relax%
    \advance\myothercount by1\relax%
    \tikzgraphsset{new <-=
      {\csname left shore prefix\the\myothercount\endcsname}
      {\csname right shore prefix\the\mycount\endcsname}{}{}}
  \ifnum\myothercount<\leftshorecount\relax
  \repeat
}
\begin{tikzpicture}
  \graph [color class=left shore, color class=right shore]
  { [operator=\zigzag]
    { [left shore, Cartesian placement]                    a, b, c },
    { [right shore, Cartesian placement, nodes={xshift=1cm}] d, e, f }
  };
\end{tikzpicture}
```

Naturally, in order to turn the above code into a usable operator, some more code would be needed (like default values and taking care of shores of different sizes).

There are a number of predefined operators, like `clique` or `cycle`, see the reference Section 19.10 for a complete list.

19.7.3 Graph Operators for Joining Groups

When you join two nodes `foo` and `bar` by the edge specification `->`, it is fairly obvious, what should happen: An edge from `(foo)` to `(bar)` should be created. However, suppose we use an edge specification between two node sets like `{a,b,c}` and `{d,e,f}`. In this case, it is not so clear which edges should be created. One might argue that all possible edges from any node in the first set to any node in the second set should be added. On the other hand, one might also argue that only a matching between these two sets should be created. Things get even more muddy when a longer chain of node sets are joined.

Instead of fixing how edges are created between two node sets, Ti*k*Z takes a somewhat more general, but also more complicated approach, which can be broken into two parts. In the following, assume that the following chain specification is given:

$\langle spec_1 \rangle$ $\langle$*edge specification*$\rangle$ $\langle spec_2 \rangle$

An example might be `{a,b,c} -> {d, e->f}`.

The source and target vertices. Let us start with the question of which vertices of the first node set should be connected to vertices in the second node set.

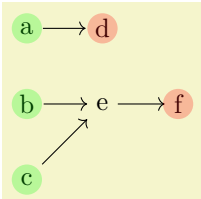
There are two predefined special color classes that are used for this: `source` and `target`. For every group specification, some vertices are colored as `source` vertices and some vertices are `target` vertices (a node can both be a target and a source). Initially, every vertex is both a source and a target, but that can change as we will see in a moment.

The intuition behind source and target vertices is that, in some sense, edges "from the outside" lead into the group via the source vertices and lead out of the group via the target vertices. To be more precise, the following happens:

1. The target vertices of the first group are connected to the source vertices of the second group.
2. In the group resulting from the union of the nodes from $\langle spec_1 \rangle$ and $\langle spec_2 \rangle$, the source vertices are only those from the first group, and the target vertices are only those from the second group.

Let us go over the effect of these rules for the example `{a,b,c} -> {d, e->f}`. First, each individual node is initially both a `source` and a `target` vertex. Then, in `{a,b,c}` all nodes are still both source and target vertices since just grouping vertices does not change their colors. Now, in `e->f` something interesting happens for the first time: the target vertices of the "group" `e` (which is just the node `e`) are connected to the source vertices of the "group" `f`. This means, that an edge is added from `e` to `f`. Then, in the resulting group `e->f` the only source vertex is `e` and the only target vertex is `f`. This implies that in the group `{d,e->f}` the sources are `d` and `e` and the targets are `d` and `f`.

Now, in `{a,b,c} -> {d,e->f}` the targets of `{a,b,c}` (which are all three of them) are connected to the sources of `{d,e->f}` (which are just `d` and `e`). Finally, in the whole graph only `a`, `b`, and `c` are sources while only `d` and `f` are targets.




```
\usetikzlibrary {graphs}
\def\hilightsource#1{\fill [green, opacity=.25] (#1) circle [radius=2mm]; }
\def\hilighttarget#1{\fill [red,   opacity=.25] (#1) circle [radius=2mm]; }
\tikz \graph
  [operator=\tikzgraphforeachcoloredmode{source}{\hilightsource},
   operator=\tikzgraphforeachcoloredmode{target}{\hilighttarget}]
  { {a,b,c} -> {d, e->f} };
```

The next objective is to make more precise what it means that "the targets of the first graph" and the "sources of the second graph" should be connected. We know already of a general way of connecting nodes of a graph: operators! Thus, we use an operator for this job. For instance, the `complete bipartite` operator adds an edge from every node having a certain color to every node have a certain other color. This is exactly what we need here: The first color is "the color `target` restricted to the nodes of the first graph" and the second color is "the color `source` restricted to the nodes of the second graph".

However, we cannot really specify that only nodes from a certain subgraph are meant – the `operator` machinery only operates on all nodes of the current graph. For this reason, what really happens is the following: When the `graph` command encounters $\langle spec_1 \rangle$ $\langle$*edge specification*$\rangle$ $\langle spec_2 \rangle$, it first computes and colors the nodes of the first and the second specification independently. Then, the `target` nodes of the first graph are recolored to `target'` and the `source` nodes of the second graph are recolored to `source'`. Then, the two graphs are united into one graph and a *joining operator* is executed, which should add edges

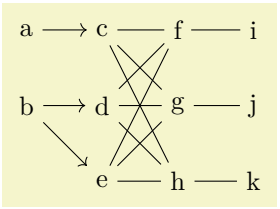
between `target'` and `source'`. Once this is done, the colors `target'` and `source'` get erased. Note that in the resulting graph only the `source` nodes from the first graph are still `source` nodes and likewise for the `target` nodes of the second graph.

The joining operators. The job of a joining operator is to add edges between nodes colored `target'` and `source'`. The following rule is used to determine which operator should be chosen for performing this job:

1. If the ⟨*edge specification*⟩ explicitly sets the `operator` key to something non-empty (and also not to `\relax`), then the ⟨*code*⟩ of this `operator` call is used.
2. Otherwise, the current value of the following key is used:

`/tikz/graphs/default edge operator=`⟨*key*⟩ (no default, initially `matching and star`)

This key stores the name of a ⟨*key*⟩ that is executed for every ⟨*edge specification*⟩ whose ⟨*options*⟩ do not contain the `operator` key.



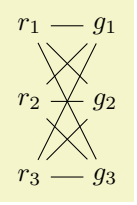

```
\usetikzlibrary {graphs}
\tikz \graph [default edge operator=matching] {
  {a, b}    ->[matching and star]
  {c, d, e} --[complete bipartite]
  {f, g, h} --
  {i, j, k}
};
```

A typical joining operator is `complete bipartite`. It takes the names of two color classes as input and adds edges from all vertices of the first class to all vertices of the second class. Now, the trick is that the default value for the `complete bipartite` key is `{target'}{source'}`. Thus, if you just write `->[complete bipartite]`, the same happens as if you had written

```
->[complete bipartite={target'}{source'}]
```

This is exactly what we want to happen. The same default values are also set for other joining operators like `matching` or `butterfly`.

Even though an operator like `complete bipartite` is typically used together with an edge specification, it can also be used as a normal operator together with a group specification. In this case, however, the color classes must be named explicitly:




```
\usetikzlibrary {graphs}
\begin{tikzpicture}
  \graph [color class=red, color class=green, math nodes]
  { [complete bipartite={red}{green}]
    { [red,   Cartesian placement]                      r_1, r_2, r_3 },
    { [green, Cartesian placement, nodes={xshift=1cm}]  g_1, g_2, g_3 }
  };
\end{tikzpicture}
```

A list of predefined joining operators can be found in the reference Section 19.10.

The fact that joining operators can also be used as normal operators leads to a subtle problem: A normal operator will typically use the current value of `default edge kind` to decide which kind of edges should be put between the identified vertices, while a joining operator should, naturally, use the kind of edge specified by the ⟨*edge specification*⟩. This problem is solved as follows: Like a normal operator, a joining operator should also use the current value of `default edge kind` for the edges it produces. The trick is that this will automatically be set to the current ⟨*edge specification*⟩ when the operator explicitly in the ⟨*options*⟩ of the edge specification or implicitly in the `default edge operator`.

19.8 Graph Macros

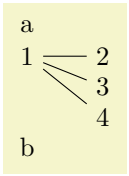
A *graph macro* is a small graph that is inserted at some point into the graph that is currently being constructed. There is special support for such graph macros in Ti*k*Z. You might wonder why this is necessary – can't one use TEX's normal macro mechanism? The answer is "no": one cannot insert new nodes into a graph using normal macros because the chains, groups, and nodes are determined prior to macro expansion.

Thus, any macro encountered where some node text should go will only be expanded when this node is being named and typeset.

A graph macro is declared using the following key:

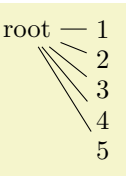
`/tikz/graphs/declare={`⟨*graph name*⟩`}{`⟨*specification*⟩`}` (no default)

This key declares that ⟨*graph name*⟩ can subsequently be used as a replacement for a ⟨*node name*⟩. Whenever the ⟨*graph name*⟩ is used in the following, a graph group will be inserted instead whose content is exactly ⟨*specification*⟩. In case ⟨*graph name*⟩ is used together with some ⟨*options*⟩, they are executed prior to inserting the ⟨*specification*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [branch down=4mm, declare={claw}{1 -- {2,3,4}}] {
  a;
  claw;
  b;
};
```

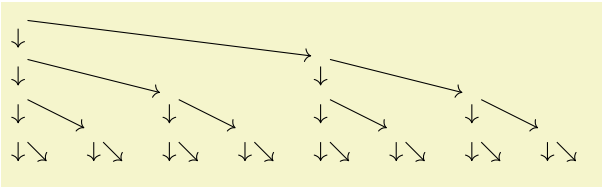
In the next example, we use a key to configure a subgraph:




```
\usetikzlibrary {graphs}
\tikz \graph [ n/.code=\def\n{#1}, branch down=4mm,
               declare={star}{root -- { \foreach \i in {1,...,\n} {\i} }}]
{ star [n=5]; };
```

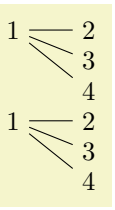
Actually, the `n` key is already defined internally for a similar purpose.

As a last example, let us define a somewhat more complicated graph macro.



```
\usetikzlibrary {graphs}
\newcount\mycount
\tikzgraphsset{
  levels/.store in=\tikzgraphlevel,
  levels=1,
  declare={bintree}{%
    [/utils/exec={%
      \ifnum\tikzgraphlevel=1\relax%
        \def\childtrees{ / }%
      \else%
        \mycount=\tikzgraphlevel%
        \advance\mycount by-1\relax%
        \edef\childtrees{
          / -> {
            bintree[levels=\the\mycount],
            bintree[levels=\the\mycount]
          }}
      \fi%
    },
    parse/.expand once=\childtrees
    ]
    % Everything is inside the \childtrees...
  }
}
\tikz \graph [grow down=5mm, branch right=5mm] { bintree [levels=5] };
```

Note that when you use a graph macro several time inside the same graph, you will typically have to use the `name` option so that different copies of the subgraph are created:




```
\usetikzlibrary {graphs}
\tikz \graph [branch down=4mm, declare={claw}{1 -- {2,3,4}}] {
  claw [name=left],
  claw [name=right]
};
```

You will find a list of useful graph macros in the reference section, Section 19.10.1.

19.9 Online Placement Strategies

The main job of the `graphs` library is to make it easy to specify which nodes are present in a graph and how they are connected. In contrast, it is *not* the primary job of the library to compute good positions for nodes in a graph – use for instance a `\matrix`, specify good positions "by hand" or use the graph drawing facilities. Nevertheless, some basic support for automatic node placement is provided for simple cases. The `graphs` library will provide you with information about the position of nodes inside their groups and chains.

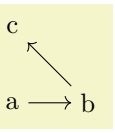
As a graph is being constructed, a *placement strategy* is used to determine a (reasonably good) position for the nodes as they are created. These placement strategies get some information about what Ti*k*Z has already seen concerning the already constructed nodes, but it gets no information concerning the upcoming nodes. Because of this lack of information concerning the future, the strategies need to be what is called an *online strategy* in computer science. (The opposite are *offline strategies*, which get information about the whole graph and all the sizes of the nodes in it. The graph drawing libraries employ such offline strategies.)

Strategies are selected using keys like `no placement` or `Cartesian placement`. It is permissible to use different strategies inside different parts of a graph, even though the different strategies do not always work together in perfect harmony.

19.9.1 Manual Placement

`/tikz/graphs/no placement` (no value)

This strategy simply "switches off" the whole placement mechanism, causing all nodes to be placed at the origin by default. You need to use this strategy if you position nodes "by hand". For this, you can use the `at` key, the `shift` keys:

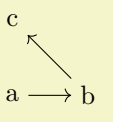



```
\usetikzlibrary {graphs}
\tikz \graph [no placement]
{
  a[at={(0:0)}] -> b[at={(1,0)}] -> c[yshift=1cm];
};
```

Since the syntax and the many braces and parentheses are a bit cumbersome, the following two keys might also be useful:

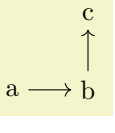
`/tikz/graphs/x=`⟨*x dimension*⟩ (no default)

When you use this key, it will have the same effect as if you had written `at={(`⟨*x dimension*⟩`,`⟨*y dimension*⟩`)}`, where ⟨*y dimension*⟩ is a value set using the `y` key:




```
\usetikzlibrary {graphs}
\tikz \graph [no placement]
{
  a[x=0,y=0] -> b[x=1,y=0] -> c[x=0,y=1];
};
```

Note that you can specify an `x` or a `y` key for a whole scope and then vary only the other key:




```
\usetikzlibrary {graphs}
\tikz \graph [no placement]
{
  a ->
  { [x=1] % group option
    b [y=0] -> c[y=1]
  };
};
```

Note that these keys have the path `/tikz/graphs/`, so they will be available inside `graph`s and will not clash with the usual `x` and `y` keys of Ti*k*Z, which are used to specify the basic lengths of vectors.

`/tikz/graphs/y=`⟨*y dimension*⟩ (no default)

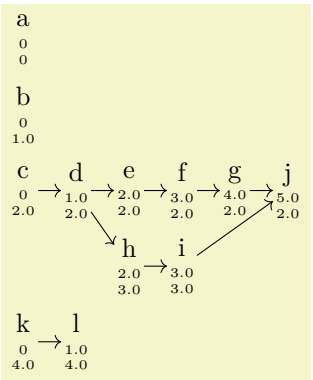
See above.

19.9.2 Placement on a Grid

`/tikz/graphs/Cartesian placement` (no value)

This strategy is the default strategy. It works, roughly, as follows: For each new node on a chain, advance a "logical width" counter and for each new node in a group, advance a "logical depth" counter. When a chain contains a whole group, then the "logical width" taken up by the group is the maximum over the logical widths taken up by the chains inside the group; and symmetrically the logical depth of a chain is the maximum of the depths of the groups inside it.

This slightly confusing explanation is perhaps best exemplified. In the below example, the two numbers indicate the two logical width and depth of each node as computed by the `graphs` library. Just ignore the arcane code that is used to print these numbers.



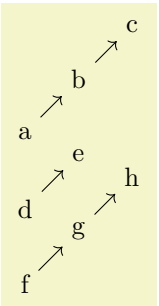

```
\usetikzlibrary {graphs}
\tikz
  \graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
          typeset={\tikzgraphnodetext\\[-4pt]
                   \tiny\mywidth\\[-6pt]\tiny\mydepth},
          placement/compute position/.append code=
            \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
            \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
{
  a,
  b,
  c -> d -> {
    e -> f -> g,
    h -> i
  } -> j,
  k -> l
};
```

You will find a detailed description of how these logical units are computed, exactly, in Section 19.9.6.

Now, even though we talk about "widths" and "depths" and even though by default a graph "grows" to the right and down, this is by no means fixed. Instead, you can use the following keys to change how widths and heights are interpreted:

`/tikz/graphs/chain shift=`⟨*coordinate*⟩ (no default, initially `(1,0)`)

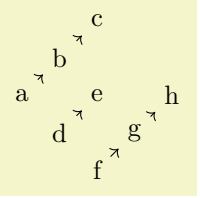
Under the regime of the `Cartesian placement` strategy, each node is shifted by the current logical width times this ⟨*coordinate*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [chain shift=(45:1)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

`/tikz/graphs/group shift=`⟨*coordinate*⟩ (no default, initially `(0,-1)`)

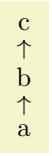
Like for `chain shift`, each node is shifted by the current logical depth times this ⟨*coordinate*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [chain shift=(45:7mm), group shift=(-45:7mm)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```

`/tikz/graphs/grow up=`⟨*distance*⟩ (default `1`)

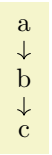
Sets the `chain shift` to `(0,`⟨*distance*⟩`)`, so that chains "grow upward". The distance by which the center of each new element is removed from the center of the previous one is ⟨*distance*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [grow up=7mm] { a -> b -> c};
```

`/tikz/graphs/grow down=`⟨*distance*⟩ (default `1`)

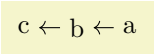
Like `grow up`.




```
\usetikzlibrary {graphs}
\tikz \graph [grow down=7mm] { a -> b -> c};
```

`/tikz/graphs/grow left=`⟨*distance*⟩ (default `1`)

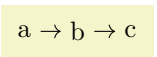
Like `grow up`.




```
\usetikzlibrary {graphs}
\tikz \graph [grow left=7mm] { a -> b -> c};
```

`/tikz/graphs/grow right=`⟨*distance*⟩ (default `1`)

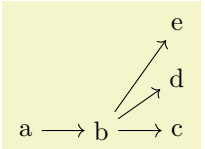
Like `grow up`.




```
\usetikzlibrary {graphs}
\tikz \graph [grow right=7mm] { a -> b -> c};
```

`/tikz/graphs/branch up=`⟨*distance*⟩ (default `1`)

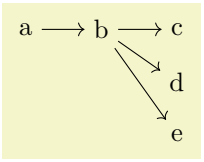
Sets the `group shift` so that groups "branch upward". The distance by which the center of each new element is removed from the center of the previous one is ⟨*distance*⟩.




```
\usetikzlibrary {graphs}
\tikz \graph [branch up=7mm] { a -> b -> {c, d, e} };
```

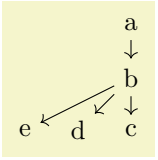
Note that when you draw a tree, the `branch ...` keys specify how siblings (or adjacent branches) are arranged, while the `grow ...` keys specify in which direction the branches "grow".

/tikz/graphs/branch down=⟨*distance*⟩ (default 1)



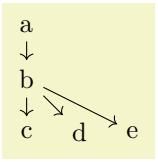

```
\usetikzlibrary {graphs}
\tikz \graph [branch down=7mm] { a -> b -> {c, d, e}};
```

/tikz/graphs/branch left=⟨*distance*⟩ (default 1)




```
\usetikzlibrary {graphs}
\tikz \graph [branch left=7mm, grow down=7mm] { a -> b -> {c, d, e}};
```

/tikz/graphs/branch right=⟨*distance*⟩ (default 1)

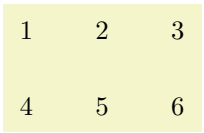



```
\usetikzlibrary {graphs}
\tikz \graph [branch right=7mm, grow down=7mm] { a -> b -> {c, d, e}};
```

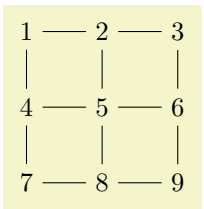
The following keys place nodes in a $N \times M$ grid.

/tikz/graphs/grid placement (no value)

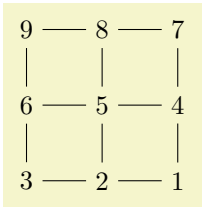
This key works similar to `Cartesian placement`. As for that placement strategy, a node has logical width and depth 1. However, the computed total width and depth are mapped to a $N \times M$ grid. The values of N and M depend on the size of the graph and the value of `wrap after`. The number of columns M is either set to `wrap after` explicitly or computed automatically as $\sqrt{|V|}$. N is the number of rows needed to lay out the graph in a grid with M columns.




```
\usetikzlibrary {graphs.standard}
% An example with 6 nodes, 3 columns and therefore 2 rows
\tikz \graph [grid placement] { subgraph I_n[n=6, wrap after=3] };
```

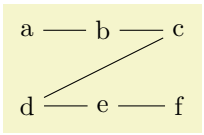



```
\usetikzlibrary {graphs.standard}
% An example with 9 nodes with columns and rows computed automatically
\tikz \graph [grid placement] { subgraph Grid_n [n=9] };
```




```
\usetikzlibrary {graphs.standard}
% Directions can be changed
\tikz \graph [grid placement, branch up, grow left] { subgraph Grid_n [n=9] };
```

In case a user-defined graph instead of a pre-defined `subgraph` is to be laid out using `grid placement`, `n` has to be specified explicitly:



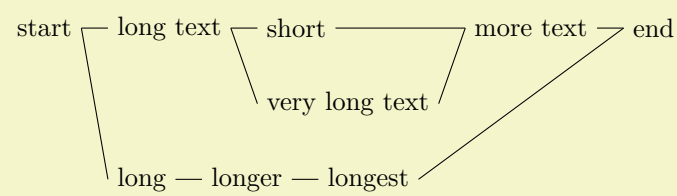

```
\usetikzlibrary {graphs}
\tikz \graph [grid placement] {
  [n=6, wrap after=3]
  a -- b -- c -- d -- e -- f
};
```

19.9.3 Placement Taking Node Sizes Into Account

Options like `grow up` or `branch right` do not take the sizes of the to-be-positioned nodes into account – all nodes are placed quite "dumbly" at grid positions. It turns out that the `Cartesian placement` can also be used to place nodes in such a way that their height and/or width is taken into account. Note, however, that while the following options may yield an adequate placement in many situations, when you need advanced alignments you should use a `matrix` or advanced offline strategies to place the nodes.

`/tikz/graphs/grow right sep=`⟨*distance*⟩ (default `1em`)

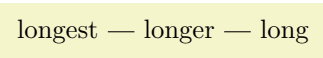
This key has several effects, but let us start with the bottom line: Nodes along a chain are placed in such a way that the left end of a new node is ⟨*distance*⟩ from the right end of the previous node:




```
\usetikzlibrary {graphs}
\tikz \graph [grow right sep, left anchor=east, right anchor=west] {
  start -- {
    long text -- {short, very long text} -- more text,
    long -- longer -- longest
  } -- end
};
```

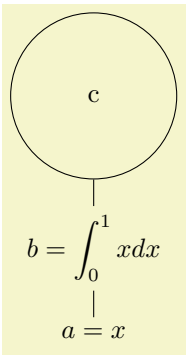
What happens internally is the following: First, the `anchor` of the nodes is set to `west` (or `north west` or `south west`, see below). Second, the logical width of a node is no longer `1`, but set to the actual width of the node (which we define as the horizontal difference between the `west` anchor and the `east` anchor) in points. Third, the `chain shift` is set to `(1pt,0pt)`.

`/tikz/graphs/grow left sep=`⟨*distance*⟩ (default `1em`)




```
\usetikzlibrary {graphs}
\tikz \graph [grow left sep] { long -- longer -- longest };
```

`/tikz/graphs/grow up sep=`⟨*distance*⟩ (default `1em`)



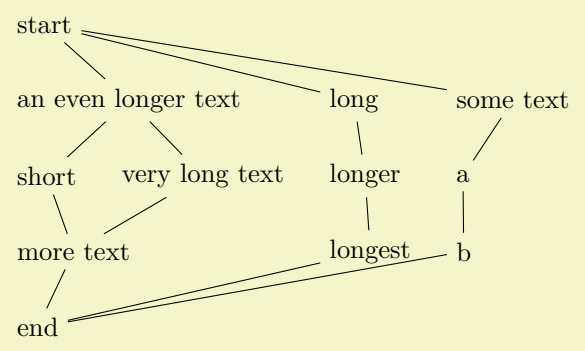

```
\usetikzlibrary {graphs}
\tikz \graph [grow up sep] {
  a / $a=x$ --
  b / {$b=\displaystyle \int_0^1 x dx$} --
  c [draw, circle, inner sep=7mm]
};
```

`/tikz/graphs/grow down sep=`⟨*distance*⟩ (default `1em`)

As above.

`/tikz/graphs/branch right sep=`⟨*distance*⟩ (default `1em`)

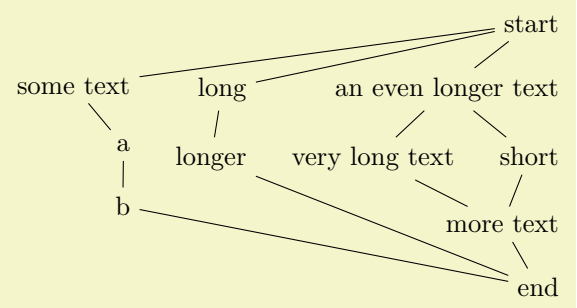
This key works like `grow right sep`, only it affects groups rather than chains.




```
\usetikzlibrary {graphs}
\tikz \graph [grow down, branch right sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer -- longest,
    some text -- a -- b
  } -- end
};
```

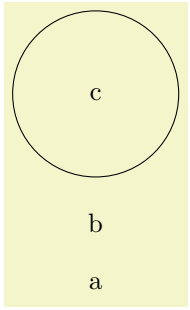
When both this key and, say, `grow down sep` are set, instead of the `west` anchor, the `north west` anchor will be selected automatically.

`/tikz/graphs/branch left sep=`⟨*distance*⟩ (default `1em`)



```
\usetikzlibrary {graphs}
\tikz \graph [grow down sep, branch left sep] {
  start -- {
    an even longer text -- {short, very long text} -- more text,
    long -- longer,
    some text -- a -- b
  } -- end
};
```

/tikz/graphs/branch up sep=⟨*distance*⟩ (default 1em)




```
\usetikzlibrary {graphs}
\tikz \graph [branch up sep] { a, b, c[draw, circle, inner sep=7mm] };
```

/tikz/graphs/branch down sep=⟨*distance*⟩ (default 1em)

19.9.4 Placement On a Circle

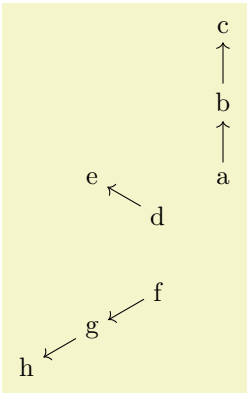
The following keys place nodes on circles. Note that, typically, you do not use `circular placement` directly, but rather use one of the two keys `clockwise` or `counterclockwise`.

/tikz/graphs/circular placement (no value)

This key works quite similar to `Cartesian placement`. As for that placement strategy, a node has logical width and depth `1`. However, the computed total width and depth are mapped to polar coordinates rather than Cartesian coordinates.

/tikz/graphs/chain polar shift=(⟨*angle*⟩:⟨*radius*⟩) (no default, initially `(0:1)`)

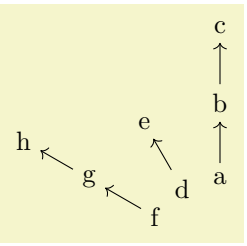
Under the regime of the `circular placement` strategy, each node on a chain is shifted by (⟨*logical width*⟩⟨*angle*⟩:⟨*logical width*⟩⟨*angle*⟩).



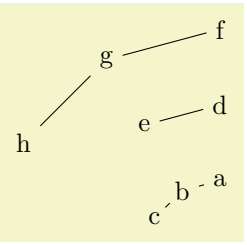

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement] {
  a -> b -> c;
  d -> e;
  f ->  g -> h;
};
```

/tikz/graphs/group polar shift=(⟨*angle*⟩:⟨*radius*⟩) (no default, initially `(45:0)`)

Like for `group shift`, each node on a chain is shifted by (⟨*logical depth*⟩⟨*angle*⟩:⟨*logical depth*⟩⟨*angle*⟩).



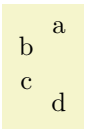

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement, group polar shift=(30:0)] {
  a -> b -> c;
  d -> e;
  f -> g -> h;
};
```



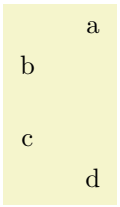

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement,
              chain polar shift=(30:0),
              group polar shift=(0:1cm)] {
  a -- b -- c;
  d -- e;
  f -- g -- h;
};
```

`/tikz/graphs/radius`=⟨*dimension*⟩ (no default, initially `1cm`)

This is an initial value that is added to the total computed radius when the polar shift of a node has been calculated. Essentially, this key allows you to set the ⟨*radius*⟩ of the innermost circle.



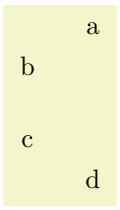

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement, radius=5mm] { a, b, c, d };
```



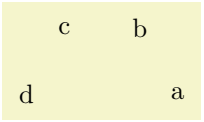

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement, radius=1cm] { a, b, c, d };
```

`/tikz/graphs/phase`=⟨*angle*⟩ (no default, initially `90`)

This is an initial value that is added to the total computed angle when the polar shift of a node has been calculated.



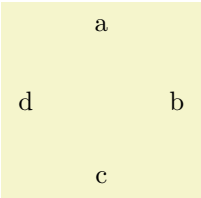

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement] { a, b, c, d };
```



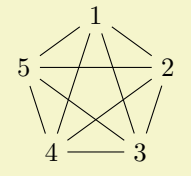

```
\usetikzlibrary {graphs}
\tikz \graph [circular placement, phase=0] { a, b, c, d };
```

/tikz/graphs/clockwise=⟨*number*⟩ (default \tikzgraphVnum)

This key sets the `group shift` so that if there are exactly ⟨*number*⟩ many nodes in a group, they will form a complete circle. If you do not provide a ⟨*number*⟩, the current value of `\tikzgraphVnum` is used, which is exactly what you want when you use predefined graph macros like `subgraph K_n`.




```
\usetikzlibrary {graphs}
\tikz \graph [clockwise=4] { a, b, c, d };
```




```
\usetikzlibrary {graphs.standard}
\tikz \graph [clockwise] { subgraph K_n [n=5] };
```

/tikz/graphs/counterclockwise=⟨*number*⟩ (default \tikzgraphVnum)

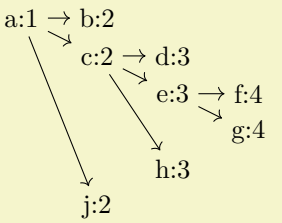
Works like `clockwise`, only the direction is inverted.

19.9.5 Levels and Level Styles

As a graph is being parsed, the `graph` command keeps track of a parameter called the *level* of a node. Provided that the graph is actually constructed in a tree-like manner, the level is exactly equal to the level of the node inside this tree.

/tikz/graphs/placement/level (no value)

This key stores a number that is increased for each element on a chain, but gets reset at the end of a group:




```
\usetikzlibrary {graphs}
\tikz \graph [ branch down=5mm, typeset=
    \tikzgraphnodetext:\pgfkeysvalueof{/tikz/graphs/placement/level}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```

Unlike the parameters `depth` and `width` described in the next section, the key `level` is always available.

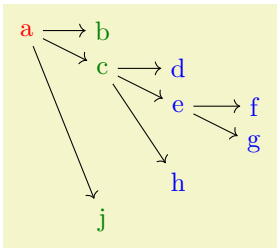
In addition to keeping track of the value of the `level` key, the `graph` command also executes the following keys whenever it creates a node:

/tikz/graph/level=⟨*level*⟩ (style, no default)

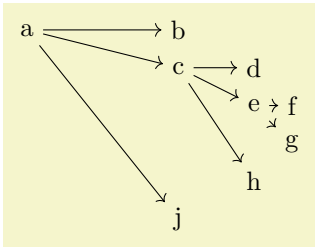
This key gets executed for each newly created node with ⟨*level*⟩ set to the current level of the node. You can use this key to, say, reconfigure the node distance or the node color.

/tikz/graph/level ⟨*level*⟩ (style, no value)

This key also gets executed for each newly created node with ⟨*level*⟩ set to the current level of the node.




```
\usetikzlibrary {graphs}
\tikz \graph [
  branch down=5mm,
  level 1/.style={nodes=red},
  level 2/.style={nodes=green!50!black},
  level 3/.style={nodes=blue}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```




```
\usetikzlibrary {graphs}
\tikz \graph [
  branch down=5mm,
  level 1/.style={grow right=2cm},
  level 2/.style={grow right=1cm},
  level 3/.style={grow right=5mm}]
{
  a -> {
    b,
    c -> {
      d,
      e -> {f,g},
      h
    },
    j
  }
};
```

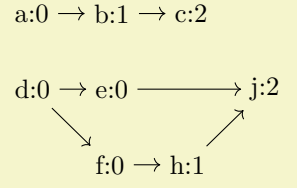
19.9.6 Defining New Online Placement Strategies

In the following the details of how to define a new placement strategy are explained. Most readers may wish to skip this section.

As a graph specification is being parsed, the `graphs` library will keep track of different numbers that identify the positions of the nodes. Let us start with what happens on a chain. First, the following counter is increased for each element of the chain:

`/tikz/graphs/placement/element count` (no value)

This key stores a number that tells us the position of the node on the current chain. However, you only have access to this value inside the code passed to the macro `compute position`, explained later on.




```
\usetikzlibrary {graphs}
\tikz \graph [
  grow right sep, typeset=\tikzgraphnodetext:\mynum,
  placement/compute position/.append code=
    \pgfkeysgetvalue{/tikz/graphs/placement/element count}{\mynum}]
{
  a -> b -> c,
  d -> {e, f->h} -> j
};
```

As can be seen, each group resets the element counter.

The second value that is computed is more complicated to explain, but it also gives more interesting information:

`/tikz/graphs/placement/width` (no value)

This key stores the "logical width" of the nodes parsed up to now in the current group or chain (more precisely, parsed since the last call of `place` in an enclosing group). This is not necessarily the "total

physical width" of the nodes, but rather a number representing how "big" the elements prior to the current element were. This *may* be their width, but it may also be their height or even their number (which, incidentally, is the default). You can use the `width` to perform shifts or rotations of to-be-created nodes (to be explained later).

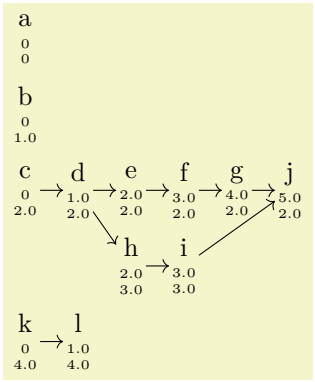
The logical width is defined recursively as follows. First, the width of a single node is computed by calling the following key:

`/tikz/graphs/placement/logical node width`=⟨*full node name*⟩ (no default)

This key is called to compute a physical or logical width of the node ⟨*full node name*⟩. You can change the code of this key. The code should return the computed value in the macro `\pgfmathresult`. By default, this key returns `1`.

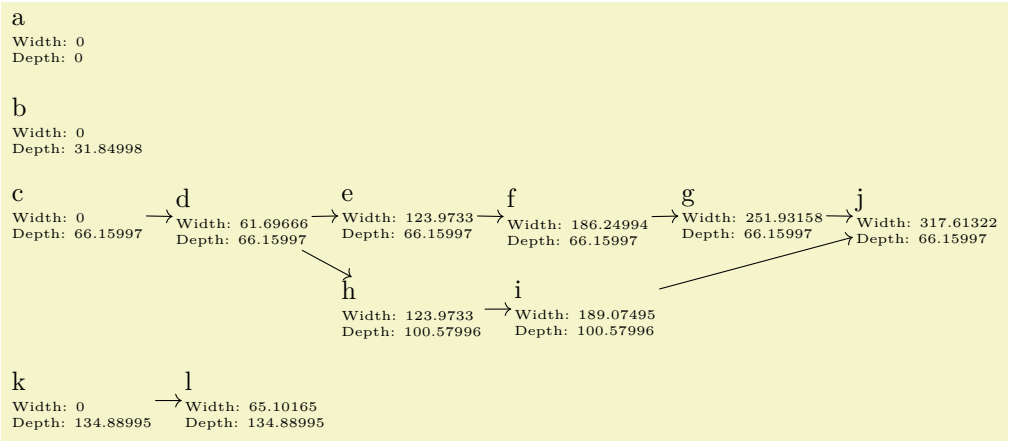
The width of a chain is the sum of the widths of its elements. The width of a group is the maximum of the widths of its elements.

To get a feeling what the above rules imply in practice, let us first have a look at an example where each node has logical width and height `1` (which is the default). The arcane options at the beginning of the code just setup things so that the computed width and depth of each node is displayed at the bottom of each node.




```
\usetikzlibrary {graphs}
\tikz
  \graph [nodes={align=center, inner sep=1pt}, grow right=7mm,
          typeset={\tikzgraphnodetext\\[-4pt]
                   \tiny\mywidth\\[-6pt]\tiny\mydepth},
          placement/compute position/.append code=
            \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
            \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
{
  a,
  b,
  c -> d -> {
    e -> f -> g,
    h -> i
  } -> j,
  k -> l
};
```

In the next example the "logical" width and depth actually match the "physical" width and height. This is caused by the `grow right sep` option, which internally sets the `logical node width` key so that it returns the width of its parameter in points.

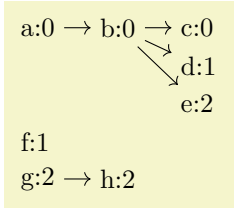


```
\usetikzlibrary {graphs}
\tikz
  \graph [grow right sep, branch down sep, nodes={align=left, inner sep=1pt},
          typeset={\tikzgraphnodetext\\[-4pt] \tiny Width: \mywidth\\[-6pt] \tiny Depth: \mydepth},
          placement/compute position/.append code=
            \pgfkeysgetvalue{/tikz/graphs/placement/width}{\mywidth}
            \pgfkeysgetvalue{/tikz/graphs/placement/depth}{\mydepth}]
{
  a,
  b,
  c -> d -> {
    e -> f -> g,
    h -> i
  } -> j,
  k -> l
};
```

Symmetrically to chains, as a group is being constructed, counters are available for the number of chains encountered so far in the current group and for the logical depth of the current group:

`/tikz/graphs/placement/chain count` (no value)

This key stores a number that tells us the sequence number of the chain in the current group.




```
\usetikzlibrary {graphs}
\tikz \graph [
  grow right sep, branch down=5mm, typeset=\tikzgraphnodetext:\mynum,
  placement/compute position/.append code=
    \pgfkeysgetvalue{/tikz/graphs/placement/chain count}{\mynum}]
{
  a -> b -> {c,d,e},
  f,
  g -> h
};
```

`/tikz/graphs/placement/depth` (no value)

Similarly to the `width` key, this key stores the "logical depth" of the nodes parsed up to now in the current group or chain and, also similarly, this key may or may not be related to the actual depth/height of the current node. As for the `width`, the exact definition is as follows: For a single node, the depth is computed by the following key:

`/tikz/graphs/placement/logical node depth=`⟨*full node name*⟩ (no default)

The code behind this key should return the "logical height" of the node ⟨*full node name*⟩ in the macro `\pgfmathresult`.

Second, the depth of a group is the sum of the depths of its elements. Third, the depth of a chain is the maximum of the depth of its elements.

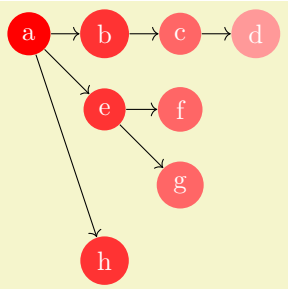
The `width`, `depth`, `element count`, and `chain count` keys get updated automatically, but do not have an effect by themselves. This is to the following two keys:

`/tikz/graphs/placement/compute position=`⟨*code*⟩ (no default)

The ⟨*code*⟩ is called by the `graph` command just prior to creating a new node (the exact moment when this key is called is detailed in the description of the `place` key). When the ⟨*code*⟩ is called, all of the keys described above will hold numbers computed in the way described above.

The job of the ⟨*code*⟩ is to setup node options appropriately so that the to-be-created node will be placed correctly. Thus, the ⟨*code*⟩ should typically set the key `nodes={shift=`⟨*coordinate*⟩`}` where ⟨*coordinate*⟩ is the computed position for the node. The ⟨*code*⟩ could also set other options like, say, the color of a node depending on its depth.

The following example appends some code to the standard code of `compute position` so that "deeper" nodes of a tree are lighter. (Naturally, the same effect could be achieved much more easily using the `level` key.)




```
\usetikzlibrary {graphs}
\newcount\mycount
\def\lightendeepernodes{
  \pgfmathsetcount{\mycount}{
    100-20*\pgfkeysvalueof{/tikz/graphs/placement/width}
  }
  \edef\mydepth{\the\mycount}
  \tikzset{nodes={fill=red!\mydepth,circle,text=white}}
}
\tikz
  \graph [placement/compute position/.append code=\lightendeepernodes]
   {
     a -> {
       b -> c -> d,
       e -> {
         f,
         g
       },
       h
     }
   };
```

/tikz/graphs/placement/place (no value)

Executing this key has two effects: First, the key `compute position` is called to compute a good position for future nodes (usually, these "future nodes" are just a single node that is created immediately). Second, all of the above counters like `depth` or `width` are reset (but not `level`).

There are two places where this key is sensibly called: First, just prior to creating a node, which happens automatically. Second, when you change the online strategy. In this case, the computed width and depth values from one strategy typically make no sense in the other strategy, which is why the new strategy should proceed "from a fresh start". In this case, the implicit call of `compute position` ensures that the new strategy gets the last place the old strategy would have used as its starting point, while the computation of its positions is now relative to this new starting point.

For these reasons, when an online strategy like `Cartesian placement` is called, this key gets called implicitly. You will rarely need to call this key directly, except when you define a new online strategy.

19.10 Reference: Predefined Elements

19.10.1 Graph Macros

TikZ Library graphs.standard

```
\usetikzlibrary{graphs.standard} % LaTeX and plain TeX
\usetikzlibrary[graphs.standard] % ConTeXt
```

This library defines a number of graph macros that are often used in the literature. When new graphs are added to this collection, they will follow the definitions in the Mathematica program, see `mathworld.wolfram.com/topics/SimpleGraphs.html`.

Graph subgraph I_n

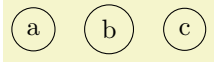
This graph consists just of n unconnected vertices. The following key is used to specify the set of these vertices:

/tikz/graphs/V={⟨*list of vertices*⟩} (no default)

Sets a list of vertex names for use with graphs like `subgraph I_n` and also other graphs. This list is available in the macro `\tikzgraphV`. The number of elements of this list is available in `\tikzgraphVnum`.

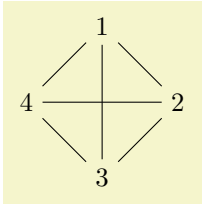
/tikz/graphs/n=⟨*number*⟩ (no default)

This is an abbreviation for `V={1,...,`⟨*number*⟩`}, name shore V/.style={name=V}`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [branch right, nodes={draw, circle}]
  { subgraph I_n [V={a,b,c}] };
```

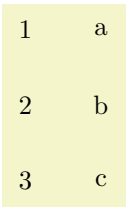
This graph is not particularly exciting by itself. However, it is often used to introduce nodes into a graph that are then connected as in the following example:




```
\usetikzlibrary {graphs.standard}
\tikz \graph [clockwise, clique] { subgraph I_n [n=4] };
```

Graph subgraph I_nm

This graph consists of two sets of once n unconnected vertices and then m unconnected vertices. The first set consists of the vertices set by the key `V`, the other set consists of the vertices set by the key `W`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph { subgraph I_nm [V={1,2,3}, W={a,b,c}] };
```

In order to set the graph path name of the two sets, the following keys get executed:

`/tikz/graphs/name shore V` (style, initially empty)

Set this style to, say, `name=my V set` in order to set a name for the `V` set.

`/tikz/graphs/name shore W` (style, initially empty)

Same as for `name shore V`.

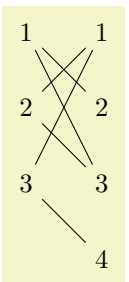
`/tikz/graphs/W={`⟨*list of vertices*⟩`}` (no default)

Sets the list of vertices for the `W` set. The elements and their number are available in the macros `\tikzgraphW` and `\tikzgraphWnum`, respectively.

`/tikz/graphs/m=`⟨*number*⟩ (no default)

This is an abbreviation for `W={1,...,`⟨*number*⟩`}, name shore W/.style={name=W}`.

The main purpose of this subgraph is to setup the nodes in a bipartite graph:

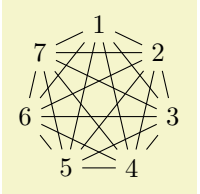



```
\usetikzlibrary {graphs.standard}
\tikz \graph {
  subgraph I_nm [n=3, m=4];

  V 1 -- { W 2, W 3 };
  V 2 -- { W 1, W 3 };
  V 3 -- { W 1, W 4 };
};
```

Graph subgraph K_n

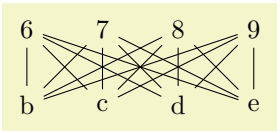
This graph is the complete clique on the vertices from the `V` key.



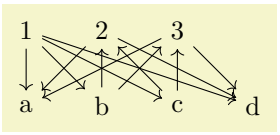

```
\usetikzlibrary {graphs.standard}
\tikz \graph [clockwise] { subgraph K_n [n=7] };
```

Graph subgraph K_nm

This graph is the complete bipartite graph with the two shores `V` and `W` as in `subgraph I_nm`.



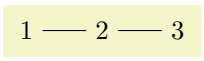

```
\usetikzlibrary {graphs.standard}
\tikz \graph [branch right, grow down]
  { subgraph K_nm [V={6,...,9}, W={b,...,e}] };
```




```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple, branch right, grow down]
{
  subgraph K_nm [V={1,2,3}, W={a,b,c,d}, ->];
  subgraph K_nm [V={2,3},   W={b,c},     <-];
};
```

Graph subgraph P_n

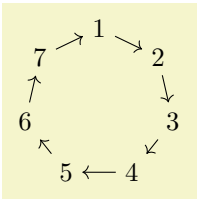
This graph is the path on the vertices in `V`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [branch right] { subgraph P_n [n=3] };
```

Graph subgraph C_n

This graph is the cycle on the vertices in `V`.



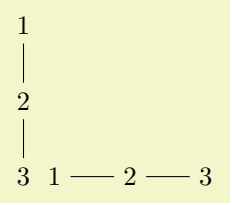

```
\usetikzlibrary {graphs.standard}
\tikz \graph [clockwise] { subgraph C_n [n=7, ->] };
```

Graph subgraph Grid_n

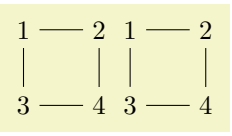
This graph is a grid of the vertices in `V`.

`/tikz/graphs/wrap after`=⟨*number*⟩ (no default)

Defines the number of nodes placed in a single row of the grid. This value implicitly defines the number of grid columns as well. In the following example a `grid placement` is used to visualize the edges created between the nodes of a `Grid_n subgraph` using different values for `wrap after`.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [grid placement] { subgraph Grid_n [n=3,wrap after=1] };
\tikz \graph [grid placement] { subgraph Grid_n [n=3,wrap after=3] };
```



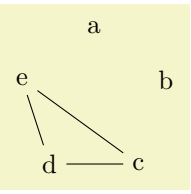

```
\usetikzlibrary {graphs.standard}
\tikz \graph [grid placement] { subgraph Grid_n [n=4,wrap after=2] };
\tikz \graph [grid placement] { subgraph Grid_n [n=4] };
```

19.10.2 Group Operators

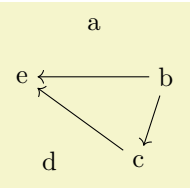
The following keys use the `operator` key to setup operators that connect the vertices of the current group having a certain color in a specific way.

`/tikz/graphs/clique=`⟨*color*⟩ (default `all`)

Adds an edge between all vertices of the current group having the (logical) color ⟨*color*⟩. Since, by default, this color is set to `all`, which is a color that all nodes get by default, when you do not specify anything, all nodes will be connected.



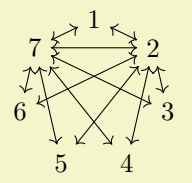

```
\usetikzlibrary {graphs}
\tikz \graph [clockwise, n=5] {
  a,
  b,
  {
    [clique]
    c, d, e
  }
};
```




```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, clockwise, n=5] {
  [clique=red, ->]
  a, b[red], c[red], d, e[red]
};
```

`/tikz/graphs/induced independent set=`⟨*color*⟩ (default `all`)

This key is the "opposite" of a `clique`: It removes all edges in the current group having belonging to color class ⟨*color*⟩. More precisely, an edge of kind `-!-` is added for each pair of vertices. This means that edge only get removed if you specify the `simple` option.

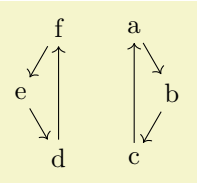



```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple] {
  subgraph K_n [<->, n=7, clockwise]; % create lots of edges

  { [induced independent set] 1, 3, 4, 5, 6 }
};
```

`/tikz/graphs/cycle=`⟨*color*⟩ (default `all`)

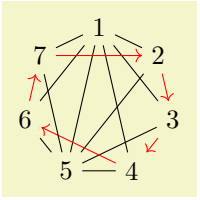
Connects the nodes colored ⟨*color*⟩ is a cyclic fashion. The ordering is the ordering in which they appear in the whole graph specification.




```
\usetikzlibrary {graphs}
\tikz \graph [clockwise, n=6, phase=60] {
  { [cycle, ->] a, b, c },
  { [cycle, <-] d, e, f }
};
```

`/tikz/graphs/induced cycle=`⟨*color*⟩ (default `all`)

While the `cycle` command will only add edges, this key will also remove all other edges between the nodes of the cycle, provided we are constructing a `simple` graph.

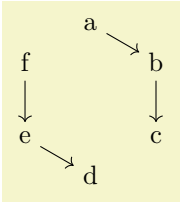



```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges

  { [induced cycle, ->, edge=red] 2, 3, 4, 6, 7 },
};
```

/tikz/graphs/path=⟨*color*⟩ (default `all`)

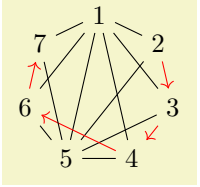
Works like `cycle`, only there is no edge from the last to the first vertex.




```
\usetikzlibrary {graphs}
\tikz \graph [clockwise, n=6] {
  { [path, ->] a, b, c },
  { [path, <-] d, e, f }
};
```

/tikz/graphs/induced path=⟨*color*⟩ (default `all`)

Works like `induced cycle`, only there is no edge from the last to the first vertex.




```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple] {
  subgraph K_n [n=7, clockwise]; % create lots of edges

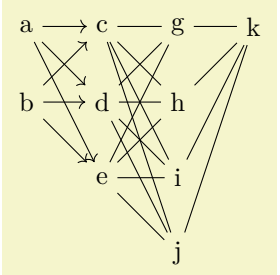
  { [induced path, ->, edges=red] 2, 3, 4, 6, 7 },
};
```

19.10.3 Joining Operators

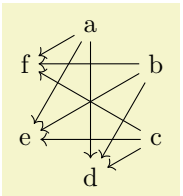
The following keys are typically used as options of an ⟨*edge specification*⟩, but can also be called in a group specification (however, then, the colors need to be set explicitly).

/tikz/graphs/complete bipartite=⟨*from color*⟩⟨*to color*⟩ (default `{source'}{target'}`)

Adds all possible edges from every node having color ⟨*from color*⟩ to every node having color ⟨*to color*⟩:



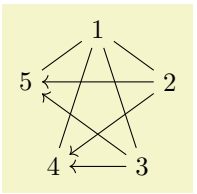

```
\usetikzlibrary {graphs}
\tikz \graph { {a, b}       ->[complete bipartite]
               {c, d, e}    --[complete bipartite]
               {g, h, i, j} --[complete bipartite]
               k };
```




```
\usetikzlibrary {graphs}
\tikz \graph [color class=red, color class=green, clockwise, n=6] {
  [complete bipartite={red}{green}, ->]
  a [red], b[red], c[red], d[green], e[green], f[green]
};
```

`/tikz/graphs/induced complete bipartite` (no value)

Works like the `complete bipartite` operator, but in a `simple` graph any edges between the vertices in either shore are removed (more precisely, they get replaced by `-!-` edges).

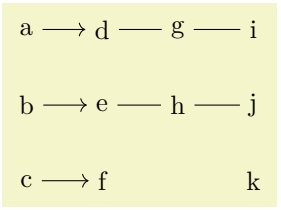



```
\usetikzlibrary {graphs.standard}
\tikz \graph [simple] {
  subgraph K_n [n=5, clockwise];  % Lots of edges

  {2, 3} ->[induced complete bipartite] {4, 5}
};
```

`/tikz/graphs/matching=`⟨*from color*⟩⟨*to color*⟩ (default `{source'}{target'}`)

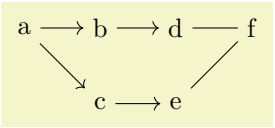
This joining operator forms a maximum *matching* between the nodes of the two sets of nodes having colors ⟨*from color*⟩ and ⟨*to color*⟩, respectively. The first node of the from set is connected to the first node of to set, the second node of the from set is connected to the second node of the to set, and so on. If the sets have the same size, what results is what graph theoreticians call a *perfect matching*, otherwise only a maximum, but not perfect matching results.




```
\usetikzlibrary {graphs}
\tikz \graph {
  {a, b, c} ->[matching]
  {d, e, f} --[matching]
  {g, h}    --[matching]
  {i, j, k}
};
```

`/tikz/graphs/matching and star=`⟨*from color*⟩⟨*to color*⟩ (default `{source'}{target'}`)

The `matching and star` connector works like the `matching` connector, only it behaves differently when the two to-be-connected sets have different size. In this case, all the surplus nodes get connected to the last node of the other set, resulting in what is known as a *star* in graph theory. This simple rule allows for some powerful effects (since this connector is the one initially set, there is no need to add it here):

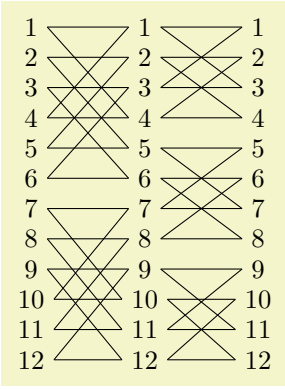



```
\usetikzlibrary {graphs}
\tikz \graph { a -> {b, c} -> {d, e} -- f};
```

The `matching and star` connector also makes it easy to create trees and series-parallel graphs.

`/tikz/graphs/butterfly=`⟨*options*⟩ (no default)

The `butterfly` connector is used to create the kind of connections present between layers of a so-called *butterfly network*. As for other connectors, two sets of nodes are connected, which are the nodes having color `target'` and `source'` by default. In a *level* l connection, the first l nodes of the first set are connected to the second l nodes of the second set, while the second l nodes of the first set get connected to the first l nodes of the second set. Then, for next $2l$ nodes of both sets a similar kind of connection is installed. Additionally, each node gets connected to the corresponding node in the other set with the same index (as in a `matching`):




```
\usetikzlibrary {graphs.standard}
\tikz \graph [left anchor=east, right anchor=west,
              branch down=4mm, grow right=15mm] {
  subgraph I_n [n=12, name=A] --[butterfly={level=3}]
  subgraph I_n [n=12, name=B] --[butterfly={level=2}]
  subgraph I_n [n=12, name=C]
};
```

Unlike most joining operators, the colors of the nodes in the first and the second set are not passed as parameters to the `butterfly` key. Rather, they can be set using the ⟨*options*⟩, which are executed with the path prefix `/tikz/graphs/butterfly`.

`/tikz/graphs/butterfly/level`=⟨*level*⟩ (no default, initially `1`)

Sets the level l for the connections.

`/tikz/graphs/butterfly/from`=⟨*color*⟩ (no default, initially `target'`)

Sets the color class of the from nodes.

`/tikz/graphs/butterfly/to`=⟨*color*⟩ (no default, initially `source'`)

Sets the color class of the to nodes.

20 Matrices and Alignment

20.1 Overview

When creating pictures, one often faces the problem of correctly aligning parts of the picture. For example, you might wish that the `baseline`s of certain nodes should be on the same line and some further nodes should be below these nodes with, say, their centers on a vertical lines. There are different ways of solving such problems. For example, by making clever use of anchors, nearly all such alignment problems can be solved. However, this often leads to complicated code. An often simpler way is to use *matrices*, the use of which is explained in the current section.

A Ti*k*Z matrix is similar to LaTeX's `{tabular}` or `{array}` environment, only instead of text each cell contains a little picture or a node. The sizes of the cells are automatically adjusted such that they are large enough to contain all the cell contents.

Matrices are a powerful tool and they need to be handled with some care. For impatient readers who skip the rest of this section: you *must* end *every* row with `\\`. In particular, the last row *must* be ended with `\\`.

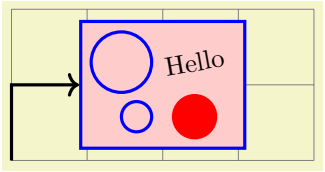
Many of the ideas implemented in Ti*k*Z's matrix support are due to Mark Wibrow – many thanks to Mark at this point!

20.2 Matrices are Nodes

Matrices are special in many ways, but for most purposes matrices are treated like nodes. This means, that you use the `node` path command to create a matrix and you only use a special option, namely the `matrix` option, to signal that the node will contain a matrix. Instead of the usual TeX-box that makes up the `text` part of the node's shape, the matrix is used. Thus, in particular, a matrix can have a shape, this shape can be drawn or filled, it can be used in a tree, and so on. Also, you can refer to the different anchors of a matrix.

`/tikz/matrix=`⟨*true or false*⟩ (default `true`)

This option can be passed to a `node` path command. It signals that the node will contain a matrix.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,2);
  \node [matrix,fill=red!20,draw=blue,very thick] (my matrix) at (2,1)
  {
    \draw (0,0)   circle [radius=4mm]; & \node[rotate=10] {Hello};            \\
    \draw (0.2,0) circle [radius=2mm]; & \fill[red] (0,0) circle [radius=3mm]; \\
  };

  \draw [very thick,->] (0,0) |- (my matrix.west);
\end{tikzpicture}
```

The exact syntax of the matrix is explained in the course of this section.

`/tikz/every matrix` (style, initially empty)

This style is used in every matrix.

`/tikz/every outer matrix` (style, initially empty)

While the `every matrix` key also applies to the matrix contents, this only applies to the outer node which holds the matrix.

Even more so than nodes, matrices will often be the only object on a path. Because of this, there is a special abbreviation for creating matrices:

`\matrix`

Inside `{tikzpicture}` this is an abbreviation for `\path node[matrix]`.

Even though matrices are nodes, some options do not have the same effect as for normal nodes:

1. Rotations and scaling have no effect on a matrix as a whole (however, you can still transform the contents of the cells normally). Before the matrix is typeset, the rotational and scaling part of the transformation matrix is reset.

2. For multi-part shapes you can only set the `text` part of the node.

3. All options starting with `text` such as `text width` have no effect.

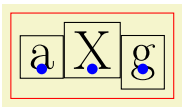
4. If you place a matrix on a path, the matrix contents will be collected into a macro, which tokenizes them. This means that `&` will lose its meaning as an alignment character, resulting in an error. If you need to place a matrix on a path, use `ampersand replacement` to work around that problem.

20.3 Cell Pictures

A matrix consists of rows of *cells*. Each row (including the last one!) is ended by the command `\\`. The character `&` is used to separate cells. Inside each cell, you must place commands for drawing a picture, called the *cell picture* in the following. (However, cell pictures are not enclosed in a complete `{pgfpicture}` environment, they are a bit more light-weight. The main difference is that cell pictures cannot have layers.) It is not necessary to specify beforehand how many rows or columns there are going to be and if a row contains less cell pictures than another line, empty cells are automatically added as needed.

20.3.1 Alignment of Cell Pictures

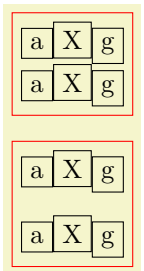
For each cell picture a bounding box is computed. These bounding boxes and the origins of the cell pictures determine how the cells are aligned. Let us start with the rows: Consider the cell pictures on the first row. Each has a bounding box and somewhere inside this bounding box the origin of the cell picture can be found (the origin might even lie outside the bounding box, but let us ignore this problem for the moment). The cell pictures are then shifted around such that all origins lie on the same horizontal line. This may make it necessary to shift some cell pictures upwards and others downwards, but it can be done and this yields the vertical alignment of the cell pictures this row. The top of the row is then given by the top of the "highest" cell picture in the row, the bottom of the row is given by the bottom of the lowest cell picture. (To be more precise, the height of the row is the maximum y-value of any of the bounding boxes and the depth of the row is the negated minimum y-value of the bounding boxes).




```
\begin{tikzpicture}
  [every node/.style={draw=black,anchor=base,font=\huge}]

  \matrix [draw=red]
  {
    \node {a}; \fill[blue] (0,0) circle [radius=2pt]; &
    \node {X}; \fill[blue] (0,0) circle [radius=2pt]; &
    \node {g}; \fill[blue] (0,0) circle [radius=2pt]; \\
  };
\end{tikzpicture}
```

Each row is aligned in this fashion: For each row the cell pictures are vertically aligned such that the origins lie on the same line. Then the second row is placed below the first row such that the bottom of the first row touches the top of the second row (unless a `row sep` is used to add a bit of space). Then the bottom of the second row touches the top of the third row, and so on. Typically, each row will have an individual height and depth.



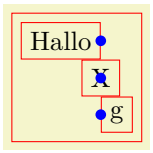

```
\begin{tikzpicture}
  [every node/.style={draw=black,anchor=base}]

  \matrix [draw=red]
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };

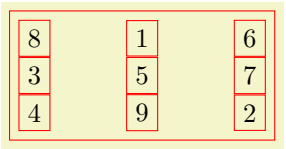
  \matrix [row sep=3mm,draw=red] at (0,-2)
  {
    \node {a}; & \node {X}; & \node {g}; \\
    \node {a}; & \node {X}; & \node {g}; \\
  };
\end{tikzpicture}
```

Let us now have a look at the columns. The rules for how the pictures on any given column are aligned are very similar to the row alignment: Consider all cell pictures in the first column. Each is shifted horizontally such that the origins lie on the same vertical line. Then, the left end of the column is at the left end of the

bounding box that protrudes furthest to the left. The right end of the column is at the right end of the bounding box that protrudes furthest to the right. This fixes the horizontal alignment of the cell pictures in the first column and the same happens the cell pictures in the other columns. Then, the right end of the first column touches the left end of the second column (unless `column sep` is used). The right end of the second column touches the left end of the third column, and so on. (Internally, two columns are actually used to achieve the desired horizontal alignment, but that is only an implementation detail.)




```
\begin{tikzpicture}[every node/.style={draw}]
  \matrix [draw=red]
  {
    \node[left]  {Hallo}; \fill[blue] (0,0) circle [radius=2pt]; \\
    \node        {X};     \fill[blue] (0,0) circle [radius=2pt]; \\
    \node[right] {g};     \fill[blue] (0,0) circle [radius=2pt]; \\
  };
\end{tikzpicture}
```



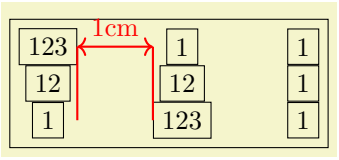

```
\begin{tikzpicture}[every node/.style={draw}]
  \matrix [draw=red,column sep=1cm]
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

20.3.2 Setting and Adjusting Column and Row Spacing

There are different ways of setting and adjusting the spacing between columns and rows. First, you can use the options `column sep` and `row sep` to set a default spacing for all rows and all columns. Second, you can add options to the `&` character and the `\\` command to adjust the spacing between two specific columns or rows. Additionally, you can specify whether the space between two columns or rows should be considered between the origins of cells in the column or row or between their borders.

`/tikz/column sep`=⟨*spacing list*⟩ (no default)

This option sets a default space that is added between every two columns. This space can be positive or negative and is zero by default. The ⟨*spacing list*⟩ normally contains a single dimension like `2pt`.

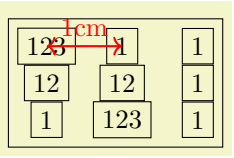



```
\begin{tikzpicture}
  \matrix [draw,column sep=1cm,nodes=draw]
  {
    \node(a) {123}; & \node (b) {1};   & \node {1}; \\
    \node    {12};  & \node     {12};  & \node {1}; \\
    \node(c) {1};   & \node (d) {123}; & \node {1}; \\
  };
  \draw [red,thick]  (a.east) -- (a.east |- c)
                     (d.west) -- (d.west |- b);
  \draw [<->,red,thick] (a.east) -- (d.west |- b)
    node [above,midway] {1cm};
\end{tikzpicture}
```

More generally, the ⟨*spacing list*⟩ may contain a whole list of numbers, separated by commas, and occurrences of the two key words `between origins` and `between borders`. The effect of specifying such a list is the following: First, all numbers occurring in the list are simply added to compute the final spacing. Second, concerning the two keywords, the last occurrence of one of the keywords is important. If the last occurrence is `between borders` or if neither occurs, then the space is inserted between the two columns normally. However, if the last occurs is `between origins`, then the following happens: The distance between the columns is adjusted such that the difference between the origins of all the

cells in the first column (remember that they all lie on straight line) and the origins of all the cells in the second column is exactly the given distance.

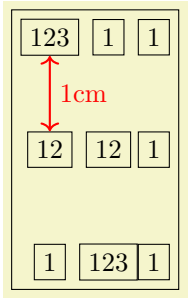
The `between origins` *option can only be used for columns mentioned in the first row, that is, you cannot specify this option for columns introduced only in later rows.*



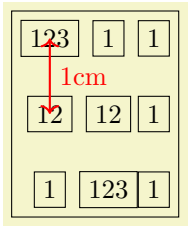

```
\begin{tikzpicture}
  \matrix [draw,column sep={1cm,between origins},nodes=draw]
  {
    \node(a) {123}; & \node (b) {1};   & \node {1}; \\
    \node    {12};  & \node     {12};  & \node {1}; \\
    \node    {1};   & \node     {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {1cm};
\end{tikzpicture}
```

`/tikz/row sep`=⟨*spacing list*⟩ (no default)

This option works like `column sep`, only for rows. Here, too, you can specify whether the space is added between the lower end of the first row and the upper end of the second row, or whether the space is computed between the origins of the two rows.

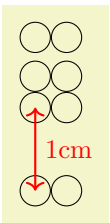



```
\begin{tikzpicture}
  \matrix [draw,row sep=1cm,nodes=draw]
  {
    \node (a) {123}; & \node {1};   & \node {1}; \\
    \node (b) {12};  & \node {12};  & \node {1}; \\
    \node     {1};   & \node {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.south) -- (b.north) node [right,midway] {1cm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,row sep={1cm,between origins},nodes=draw]
  {
    \node (a) {123}; & \node {1};   & \node {1}; \\
    \node (b) {12};  & \node {12};  & \node {1}; \\
    \node     {1};   & \node {123}; & \node {1}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

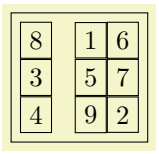
The row-end command `\\` allows you to provide an optional argument, which must be a dimension. This dimension will be added to the list in `row sep`. This means that, firstly, any numbers you list in this argument will be added as an extra row separation between the line being ended and the next line and, secondly, you can use the keywords `between origins` and `between borders` to locally overrule the standard setting for this line pair.



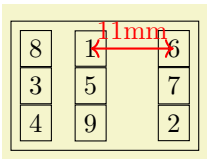

```
\begin{tikzpicture}
  \matrix [row sep=1mm, radius=2mm]
  {
    \draw (0,0) circle; & \draw (0,0) circle; \\
    \draw (0,0) circle; & \draw (0,0) circle; \\[-1mm]
    \draw (0,0) coordinate (a) circle; &
    \draw (0,0) circle; \\[1cm,between origins]
    \draw (0,0) coordinate (b) circle; &
    \draw (0,0) circle; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [right,midway] {1cm};
\end{tikzpicture}
```

The cell separation character `&` also takes an optional argument, which must also be a spacing list. This spacing list is added to the `column sep` having a similar effect as the option for the `\\` command for rows.

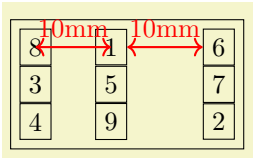
This optional spacing list can only be given the first time a new column is started (usually in the first row), subsequent usages of this option in later rows have no effect.



```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &[2mm] \node{1}; &[-1mm] \node {6}; \\
    \node {3}; &       \node{5}; &       \node {7}; \\
    \node {4}; &       \node{9}; &       \node {2}; \\
  };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep=1mm]
  {
    \node {8}; &[2mm] \node(a){1}; &[1cm,between origins] \node(b){6}; \\
    \node {3}; &       \node   {5}; &                      \node   {7}; \\
    \node {4}; &       \node   {9}; &                      \node   {2}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {11mm};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \matrix [draw,nodes=draw,column sep={1cm,between origins}]
  {
    \node (a) {8}; & \node (b) {1}; &[between borders] \node (c) {6}; \\
    \node     {3}; & \node     {5}; &                  \node     {7}; \\
    \node     {4}; & \node     {9}; &                  \node     {2}; \\
  };
  \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
  \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway] {10mm};
\end{tikzpicture}
```

20.3.3 Cell Styles and Options

The following styles and options are useful for changing the appearance of all cell pictures:

`/tikz/every cell={`⟨*row*⟩`}{`⟨*column*⟩`}` (style, no default, initially empty)

This style is installed at the beginning of each cell picture with the two parameters being the current ⟨*row*⟩ and ⟨*column*⟩ of the cell. Note that setting this style to `draw` will *not* cause all nodes to be drawn since the `draw` option has to be passed to each node individually.

Inside this style (and inside all cells), the current ⟨*row*⟩ and ⟨*column*⟩ number are also accessible via the counters `\pgfmatrixcurrentrow` and `\pgfmatrixcurrentcolumn`.

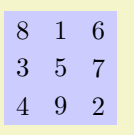
`/tikz/cells=`⟨*options*⟩ (no default)

This key adds the ⟨*options*⟩ to the style `every cell`. It is mainly just a shorthand for the code `every cell/.append style=`⟨*options*⟩.

`/tikz/nodes=`⟨*options*⟩ (no default)

This key adds the ⟨*options*⟩ to the style `every node`. It is mainly just a shorthand for the code `every node/.append style=`⟨*options*⟩.

The main use of this option is the install some options for the nodes *inside* the matrix that should not apply to the matrix *itself*.



```
\begin{tikzpicture}
  \matrix [nodes={fill=blue!20,minimum size=5mm}]
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

The next set of styles can be used to change the appearance of certain rows, columns, or cells. If more than one of these styles is defined, they are executed in the below order (the `every cell` style is executed before all of the below).

/tikz/column ⟨*number*⟩ (style, no value)

This style is used for every cell in column ⟨*number*⟩.

/tikz/every odd column (style, no value)

This style is used for every cell in an odd column.

/tikz/every even column (style, no value)

This style is used for every cell in an even column.

/tikz/row ⟨*number*⟩ (style, no value)

This style is used for every cell in row ⟨*number*⟩.

/tikz/every odd row (style, no value)

This style is used for every cell in an odd row.

/tikz/every even row (style, no value)

This style is used for every cell in an even row.

/tikz/row ⟨*row number*⟩ column ⟨*column number*⟩ (style, no value)

This style is used for the cell in row ⟨*row number*⟩ and column ⟨*column number*⟩.

8 1 6
3 5 7
4 9 2

```
\begin{tikzpicture}
  [row 1/.style={red},
   column 2/.style={green!50!black},
   row 3 column 3/.style={blue}]

  \matrix
  {
    \node {8}; & \node{1}; & \node {6}; \\
    \node {3}; & \node{5}; & \node {7}; \\
    \node {4}; & \node{9}; & \node {2}; \\
  };
\end{tikzpicture}
```

You can use the `column` ⟨*number*⟩ option to change the alignment for different columns.

123 456 789
12 45 78
1 4 7

```
\begin{tikzpicture}
  [column 1/.style={anchor=base west},
   column 2/.style={anchor=base east},
   column 3/.style={anchor=base}]
  \matrix
  {
    \node {123}; & \node{456}; & \node {789}; \\
    \node {12}; & \node{45}; & \node {78}; \\
    \node {1}; & \node{4}; & \node {7}; \\
  };
\end{tikzpicture}
```

In some cases, it is desirable to include some automation in each column/row separately. A typical example is to apply stripe-pattern to almost all columns with exceptions. For these type of use-cases, nesting these keys can open up a lot of possibilities; in the following example a "feature comparison" table is demonstrated. It is intentionally made rather verbose and a bit redundant to show how the column and row settings can be progressively overwritten to create certain effects.

	Basic	Standard	Professional	Enterprise
Feature A	•	•	•	•
Feature B	•	•	•	•
Feature C				•
Feature D		•	•	•
Feature E			•	•

Popular Choice!

```
\usetikzlibrary {matrix,fit}
\begin{tikzpicture}[
  font=\sffamily,
  head color/.style args={#1/#2}{
    row 1 column #1/.append style={nodes={fill=#2}}},
  % swap order of row and column styles
  matrix/inner style order={
    every cell,
    row, even odd row,
    column, even odd column,
    cell
  }
]

\matrix [
   matrix of nodes, nodes in empty cells,
   nodes={text width=2cm, align=center,
          minimum height=1.5em, anchor=center},
   % add striped row style
   every even row/.style={nodes={fill=olive!50}},
   % modify the feature column and header row
   column 1/.style=      {nodes={fill=olive, inner ysep=0}},
   row 1/.style=         {nodes={text depth=0.2ex, text=white}},
   row 1 column 1/.style={nodes={fill=none, draw=none}},
   head color/.list={2/orange,3/teal,4/cyan,5/magenta} % specify header colors
  ] (m)
  {
            & Basic     & Standard   & Professional & Enterprise \\
  Feature A & $\bullet$ & $\bullet$  & $\bullet$    & $\bullet$  \\
  Feature B & $\bullet$ & $\bullet$  & $\bullet$    & $\bullet$  \\
  Feature C &           &            &              & $\bullet$  \\
  Feature D &           & $\bullet$  & $\bullet$    & $\bullet$  \\
  Feature E &           &            & $\bullet$    & $\bullet$  \\
  };
% Add emphasis on selection by the use of "fit" library
\node[fit={(m-1-4.north west) (m-6-4.south east)},
      ultra thick, inner sep=0pt, rounded corners=1mm,
      draw=cyan, label={[cyan,align=center]270:Popular\\Choice!}]{};
\end{tikzpicture}
```

The order in which these styles are applied is configurable. You can also install your own styles. The following styles (in fact, internally they are `/.code` keys) wrap the styles introduced in the previous paragraph passing the correct argument and ensuring that they are only called for even or odd rows. However, it is not recommended to override these.

`/tikz/matrix/inner style/every cell` (style, no value)

Wraps `/tikz/every cell`.

`/tikz/matrix/inner style/column` (style, no value)

Wraps `/tikz/column` ⟨*number*⟩.

`/tikz/matrix/inner style/even odd column` (style, no value)

Wraps `/tikz/every even column` and `/tikz/every odd column`.

`/tikz/matrix/inner style/row` (style, no value)

Wraps `/tikz/row` ⟨*number*⟩.

`/tikz/matrix/inner style/even odd row` (style, no value)

Wraps `/tikz/every even row` and `/tikz/every odd row`.

`/tikz/matrix/inner style/cell` (style, no value)

Wraps `/tikz/row` ⟨*number*⟩ `column` ⟨*number*⟩.

`/tikz/matrix/inner style order` (style, no value)

The order in which these styles are applied to the matrix cells is specified by this key. By default it is

```
\tikzset{
  matrix/inner style order={
    every cell,
    column,
    even odd column,
    row,
    even odd row,
    cell,
  },
}
```

You can use this to install your own styles here, but only *names* of styles are permitted here. The style specification has to be placed outside of `matrix/inner style order` and unless it is installed inside `/tikz/matrix/inner style/`, it has to be fully qualified.

```
\tikzset{
  my style/.code={%
    \ifnum\pgfmatrixcurrentcolumn=2
        \tikzset{font=\itshape}%
    \fi
  },
  matrix/inner style order={
      every cell,
      even odd column,
      even odd row,
      column,
      row,
      cell,
      /tikz/my style,
  },
}
```

In many matrices all cell pictures have nearly the same code. For example, cells typically start with `\node{` and end `};`. The following options allow you to execute such code in all cells:

`/tikz/execute at begin cell=`⟨*code*⟩ (no default)

The code will be executed at the beginning of each nonempty cell.

`/tikz/execute at end cell=`⟨*code*⟩ (no default)

The code will be executed at the end of each nonempty cell.

`/tikz/execute at empty cell=`⟨*code*⟩ (no default)

The code will be executed inside each empty cell.

8	1	6
3	5	7
4	9	2

```
\begin{tikzpicture}
  [matrix of nodes/.style={
     execute at begin cell=\node\bgroup,
     execute at end cell=\egroup;%
   }]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

8	1	–
3	–	7
–	–	2

```
\begin{tikzpicture}
  [matrix of nodes/.style={
     execute at begin cell=\node\bgroup,
     execute at end cell=\egroup;,%
     execute at empty cell=\node{--};%
   }]
  \matrix [matrix of nodes]
  {
    8 & 1 &   \\
    3 &   & 7 \\
      &   & 2 \\
  };
\end{tikzpicture}
```

The `matrix` library defines a number of styles that make use of the above options.

20.4 Anchoring a Matrix

Since matrices are nodes, they can be anchored in the usual fashion using the `anchor` option. However, there are two ways to influence this placement further. First, the following option is often useful:

`/tikz/matrix anchor=`⟨*anchor*⟩ (no default)

This option has the same effect as `anchor`, but the option applies only to the matrix itself, not to the cells inside. If you just say `anchor=north` as an option to the matrix node, all nodes inside matrix will also have this anchor, unless it is explicitly set differently for each node. By comparison, `matrix anchor` sets the anchor for the matrix, but for the nodes inside the value of `anchor` remain unchanged.

123
12
1

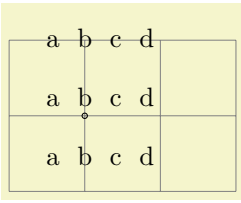
123
12
1

```
\begin{tikzpicture}
  \matrix [matrix anchor=west] at (0,0)
  {
    \node {123}; \\ % still center anchor
    \node {12}; \\
    \node {1}; \\
  };
  \matrix [anchor=west] at (0,-2)
  {
    \node {123}; \\ % inherited west anchor
    \node {12}; \\
    \node {1}; \\
  };
\end{tikzpicture}
```

The second way to anchor a matrix is to use *an anchor of a node inside the matrix*. For this, the `anchor` option has a special effect when given as an argument to a matrix:

`/tikz/anchor=`⟨*anchor or node.anchor*⟩ (no default)

Normally, the argument of this option refers to anchor of the matrix node, which is the node that includes all of the stuff of the matrix. However, you can also provide an argument of the form ⟨*node*⟩`.`⟨*anchor*⟩ where ⟨*node*⟩ must be node defined inside the matrix and ⟨*anchor*⟩ is an anchor of this node. In this case, the whole matrix is shifted around in such a way that this particular anchor of this particular node lies at the `at` position of the matrix. The same is true for `matrix anchor`.



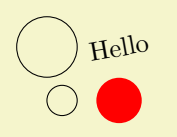

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \matrix[matrix anchor=inner node.south,anchor=base,row sep=3mm] at (1,1)
  {
    \node {a}; & \node              {b}; & \node {c}; & \node {d}; \\
    \node {a}; & \node(inner node) {b}; & \node {c}; & \node {d}; \\
    \node {a}; & \node              {b}; & \node {c}; & \node {d}; \\
  };
  \draw (inner node.south) circle [radius=1pt];
\end{tikzpicture}
```

20.5 Considerations Concerning Active Characters

Even though Ti*k*Z seems to use `&` to separate cells, PGF actually uses a different command to separate cells, namely the command `\pgfmatrixnextcell` and using a normal `&` character will normally fail. What happens is that, Ti*k*Z makes `&` an active character and then defines this character to be equal to `\pgfmatrixnextcell`. In most situations this will work nicely, but sometimes `&` cannot be made active; for instance because the matrix is used in an argument of some macro or the matrix contains nodes that contain normal `{tabular}` environments. In this case you can use the following option to avoid having to type `\pgfmatrixnextcell` each time:

`/tikz/ampersand replacement=`⟨*macro name or empty*⟩ (no default)

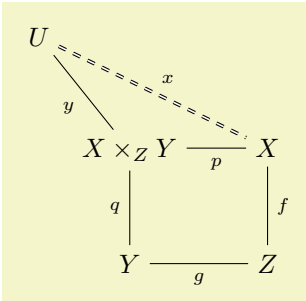
If a macro name is provided, this macro will be defined to be equal to `\pgfmatrixnextcell` inside matrices and `&` will not be made active. For instance, you could say `ampersand replacement=\&` and then use `\&` to separate columns as in the following example:



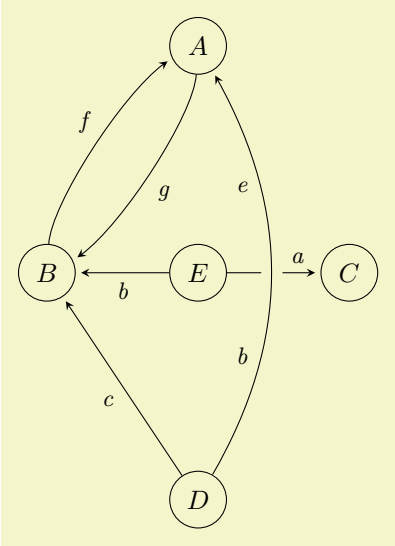

```
\tikz
  \matrix [ampersand replacement=\&]
  {
    \draw (0,0)   circle [radius=4mm]; \& \node[rotate=10] {Hello};              \\
    \draw (0.2,0) circle [radius=2mm]; \& \fill[red] (0,0) circle [radius=3mm]; \\
  };
```

20.6 Examples

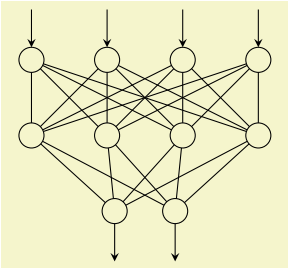
The following examples are adapted from code by Mark Wibrow. The first two redraw pictures from Timothy van Zandt's PStricks documentation:




```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,row sep=1cm]
  {
    |(U)| U &[2mm]                          &[8mm]     \\
            &      |(XZY)| X \times_Z Y  &      |(X)| X \\
            &      |(Y)|   Y              &      |(Z)| Z \\
  };
  \begin{scope}[every node/.style={midway,auto,font=\scriptsize}]
    \draw [double, dashed] (U)    -- node {$x$} (X);
    \draw                  (X)    -- node {$p$} (X -| XZY.east)
                           (X)    -- node {$f$} (Z)
                                  -- node {$g$} (Y)
                                  -- node {$q$} (XZY)
                                  -- node {$y$} (U);
   \end{scope}
\end{tikzpicture}
```

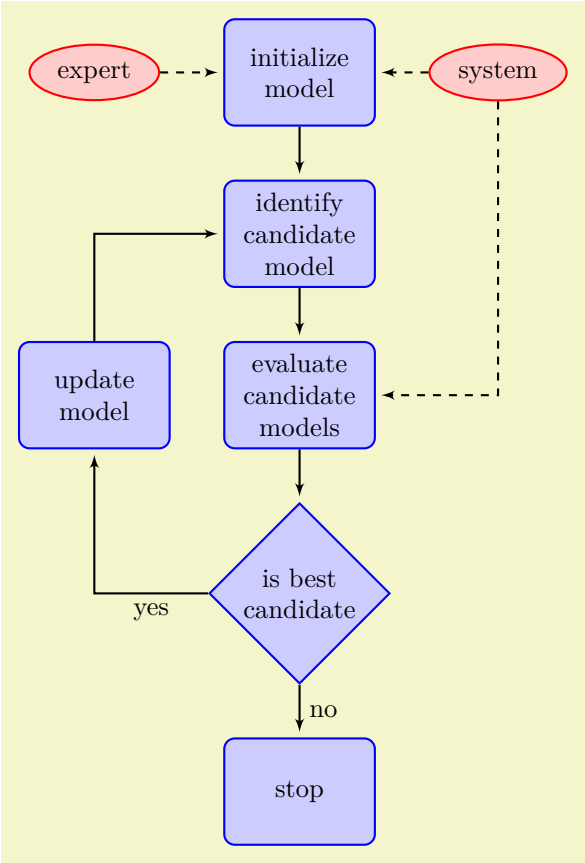


```
\usetikzlibrary {matrix}
\begin{tikzpicture}[>=stealth,->,shorten >=2pt,looseness=.5,auto]
  \matrix [matrix of math nodes,
           column sep={2cm,between origins},
           row sep={3cm,between origins},
           nodes={circle, draw, minimum size=7.5mm}]
  {
             & |(A)| A &         \\
    |(B)| B & |(E)| E & |(C)| C \\
             & |(D)| D           \\
  };
  \begin{scope}[every node/.style={font=\small\itshape}]
    \draw (A) to [bend left] node [midway]   {g} (B);
    \draw (B) to [bend left] node [midway]   {f} (A);
    \draw (D) --             node [midway]   {c} (B);
    \draw (E) --             node [midway]   {b} (B);
    \draw (E) --             node [near end] {a} (C);
    \draw [-,line width=8pt,draw=graphicbackground]
          (D) to [bend right, looseness=1] (A);
    \draw (D) to [bend right, looseness=1]
           node [near start] {b} node [near end] {e} (A);
  \end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix (network)
    [matrix of nodes,%
     nodes in empty cells,
     nodes={outer sep=0pt,circle,minimum size=4pt,draw},
     column sep={1cm,between origins},
     row sep={1cm,between origins}]
  {
                    &                 &                   & \\
                    &                 &                   & \\
    |[draw=none]| & |[xshift=1mm]| & |[xshift=-1mm]|   \\
  };
  \foreach \a in {1,...,4}{
    \draw (network-3-2) -- (network-2-\a);
    \draw (network-3-3) -- (network-2-\a);
    \draw [-stealth] ([yshift=5mm]network-1-\a.north) -- (network-1-\a);
    \foreach \b in {1,...,4}
      \draw (network-1-\a) -- (network-2-\b);
  }
  \draw [stealth-] ([yshift=-5mm]network-3-2.south) -- (network-3-2);
  \draw [stealth-] ([yshift=-5mm]network-3-3.south) -- (network-3-3);
\end{tikzpicture}
```

The following example is adapted from code written by Kjell Magne Fauske, which is based on the following paper: K. Bossley, M. Brown, and C. Harris, Neurofuzzy identification of an autonomous underwater vehicle, *International Journal of Systems Science*, 1999, 30, 901–913.


expert
initialize model
system
identify candidate model
update model
evaluate candidate models
is best candidate
yes
no
stop

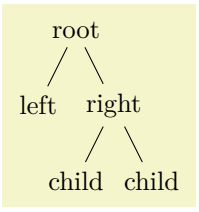
```
\usetikzlibrary {arrows,shapes.geometric}
\begin{tikzpicture}
  [auto,
   decision/.style={diamond, draw=blue, thick, fill=blue!20,
                    text width=4.5em,align=flush center,
                    inner sep=1pt},
   block/.style   ={rectangle, draw=blue, thick, fill=blue!20,
                    text width=5em,align=center, rounded corners,
                    minimum height=4em},
   line/.style    ={draw, thick, -latex',shorten >=2pt},
   cloud/.style   ={draw=red, thick, ellipse,fill=red!20,
                    minimum height=2em}]

  \matrix [column sep=5mm,row sep=7mm]
  {
    % row 1
      \node [cloud] (expert)   {expert}; &
      \node [block] (init)     {initialize model}; &
      \node [cloud] (system)   {system}; \\
    % row 2
      & \node [block] (identify) {identify candidate model}; & \\
    % row 3
      \node [block] (update)   {update model};  &
      \node [block] (evaluate) {evaluate candidate models}; & \\
    % row 4
      & \node [decision] (decide) {is best candidate}; & \\
    % row 5
      & \node [block] (stop)      {stop}; & \\
  };
  \begin{scope}[every path/.style=line]
    \path          (init)     -- (identify);
    \path          (identify) -- (evaluate);
    \path          (evaluate) -- (decide);
    \path          (update)   |- (identify);
    \path          (decide)   -| node [near start] {yes} (update);
    \path          (decide)   -- node [midway] {no} (stop);
    \path [dashed] (expert)   -- (init);
    \path [dashed] (system)   -- (init);
    \path [dashed] (system)   |- (evaluate);
  \end{scope}
\end{tikzpicture}
```

21 Making Trees Grow

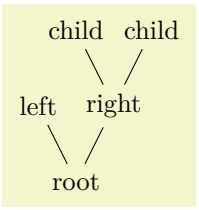
21.1 Introduction to the Child Operation

Trees are a common way of visualizing hierarchical structures. A simple tree looks like this:




```
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Admittedly, in reality trees are more likely to grow *upward* and not downward as above. You can tell whether the author of a paper is a mathematician or a computer scientist by looking at the direction their trees grow. A computer scientist's trees will grow downward while a mathematician's tree will grow upward. Naturally, the *correct* way is the mathematician's way, which can be specified as follows:




```
\begin{tikzpicture}
  \node {root} [grow'=up]
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

In Ti*k*Z, there are two ways of specifying trees: Using either the `graph` path operation, which is covered in Section 19, or using the `child` path operation, which is covered in the present section. Both methods have their advantages.

In Ti*k*Z, trees are specified by adding *children* to a node on a path using the `child` operation:

`\path` … `child[`⟨*options*⟩`]foreach`⟨*variables*⟩`in{`⟨*values*⟩`}{`⟨*child path*⟩`}` … `;`

This operation should directly follow a completed `node` operation or another `child` operation, although it is permissible that the first `child` operation is preceded by options (we will come to that).

When a `node` operation like `node {X}` is followed by `child`, Ti*k*Z starts counting the number of child nodes that follow the original `node {X}`. For this, it scans the input and stores away each `child` and its arguments until it reaches a path operation that is not a `child`. Note that this will fix the character codes of all text inside the child arguments, which means, in essence, that you cannot use verbatim text inside the nodes inside a `child`. Sorry.

Once the children have been collected and counted, Ti*k*Z starts generating the child nodes. For each child of a parent node Ti*k*Z computes an appropriate position where the child is placed. For each child, the coordinate system is transformed so that the origin is at this position. Then the ⟨*child path*⟩ is drawn. Typically, the child path just consists of a `node` specification, which results in a node being drawn at the child's position. Finally, an edge is drawn from the first node in the ⟨*child path*⟩ to the parent node.

The optional `foreach` part (note that there is no backslash before `foreach`) allows you to specify multiple children in a single `child` command. The idea is the following: A `\foreach` statement is (internally) used to iterate over the list of ⟨*values*⟩. For each value in this list, a new `child` is added to the node. The syntax for ⟨*variables*⟩ and for ⟨*values*⟩ is the same as for the `\foreach` statement, see Section 88. For example, when you say

```
node {root} child [red] foreach \name in {1,2} {node {\name}}
```

the effect will be the same as if you had said

```
node {root} child[red] {node {1}} child[red] {node {2}}
```

When you write

```
node {root} child[\pos] foreach \name/\pos in {1/left,2/right} {node[\pos] {\name}}
```

the effect will be the same as for

```
node {root} child[left] {node[left] {1}} child[right] {node[right] {2}}
```

You can nest things as in the following example:



```
\begin{tikzpicture}
  [level distance=4mm,level/.style={sibling distance=8mm/#1}]
  \coordinate
    child foreach \x in {0,1}
      {child foreach \y in {0,1}
        {child foreach \z in {0,1}}};
\end{tikzpicture}
```

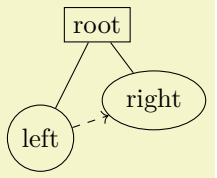
The details and options for this operation are described in the rest of this present section.

21.2 Child Paths and Child Nodes

For each `child` of a root node, its ⟨*child path*⟩ is inserted at a specific location in the picture (the placement rules are discussed in Section 21.5). The first node in the ⟨*child path*⟩, if it exists, is special and called the *child node*. If there is no first node in the ⟨*child path*⟩, that is, if the ⟨*child path*⟩ is missing (including the curly braces) or if it does not start with `node` or with `coordinate`, then an empty child node of shape `coordinate` is automatically added.

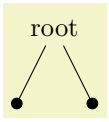
Consider the example `\node {x} child {node {y}} child;`. For the first child, the ⟨*child path*⟩ has the child node `node {y}`. For the second child, no child node is specified and, thus, it is just `coordinate`.

As for any normal node, you can give the child node a name, shift it around, or use options to influence how it is rendered.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[sibling distance=15mm]
  \node[rectangle,draw] {root}
    child {node[circle,draw,yshift=-5mm] (left node) {left}}
    child {node[ellipse,draw] (right node) {right}};
  \draw[dashed,->] (left node) -- (right node);
\end{tikzpicture}
```

In many cases, the ⟨*child path*⟩ will just consist of a specification of a child node and, possibly, children of this child node. However, the node specification may be followed by arbitrary other material that will be added to the picture, transformed to the child's coordinate system. For your convenience, a move-to `(0,0)` operation is inserted automatically at the beginning of the path. Here is an example:




```
\begin{tikzpicture}
  \node {root}
    child {[fill] circle [radius=2pt]}
    child {[fill] circle [radius=2pt]};
\end{tikzpicture}
```

At the end of the ⟨*child path*⟩ you may add a special path operation called `edge from parent`. If this operation is not given by yourself somewhere on the path, it will be automatically added at the end. This option causes a connecting edge from the parent node to the child node to be added to the path. By giving options to this operation you can influence how the edge is rendered. Also, nodes following the `edge from parent` operation will be placed on this edge, see Section 21.6 for details.

To sum up:

1. The child path starts with a node specification. If it is not there, it is added automatically.
2. The child path ends with a `edge from parent` operation, possibly followed by nodes to be put on this edge. If the operation is not given at the end, it is added automatically.

21.3 Naming Child Nodes

Child nodes can be named like any other node using either the `name` option or the special syntax in which the name of the node is placed in round parentheses between the `node` operation and the node's text.

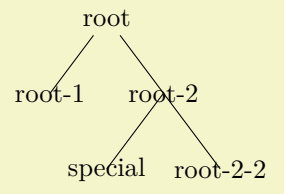
If you do not assign a name to a child node, Ti*k*Z will automatically assign a name as follows: Assume that the name of the parent node is, say, `parent`. (If you did not assign a name to the parent, Ti*k*Z will do

so itself, but that name will not be user-accessible.) The first child of `parent` will be named `parent-1`, the second child is named `parent-2`, and so on.

This naming convention works recursively. If the second child `parent-2` has children, then the first of these children will be called `parent-2-1` and the second `parent-2-2` and so on.

If you assign a name to a child node yourself, no name is generated automatically (the node does not have two names). However, "counting continues", which means that the third child of `parent` is called `parent-3` independently of whether you have assigned names to the first and/or second child of `parent`.

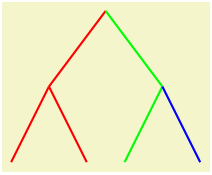
Here is an example:




```
\begin{tikzpicture}[sibling distance=15mm]
  \node (root) {root}
    child
    child {
      child {coordinate (special)}
      child
    };
  \node at (root-1) {root-1};
  \node at (root-2) {root-2};
  \node at (special) {special};
  \node at (root-2-2) {root-2-2};
\end{tikzpicture}
```

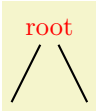
21.4 Specifying Options for Trees and Children

Each `child` may have its own ⟨*options*⟩, which apply to "the whole child", including all of its grandchildren. Here is an example:



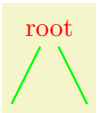
```
\begin{tikzpicture}
  [thick,level 1/.style={sibling distance=15mm},
         level 2/.style={sibling distance=10mm}]
  \coordinate
    child[red]   {child child}
    child[green] {child child[blue]};
\end{tikzpicture}
```

The options of the root node have no effect on the children since the options of a node are always "local" to that node. Because of this, the edges in the following tree are black, not red.




```
\begin{tikzpicture}[thick]
  \node [red] {root}
    child
    child;
\end{tikzpicture}
```

This raises the problem of how to set options for *all* children. Naturally, you could always set options for the whole path as in `\path [red] node {root} child child;` but this is bothersome in some situations. Instead, it is easier to give the options *before the first child* as follows:




```
\begin{tikzpicture}[thick]
  \node [red] {root}
    [green] % option applies to all children
    child
    child;
\end{tikzpicture}
```

Here is the set of rules:

1. Options for the whole tree are given before the root node.
2. Options for the root node are given directly to the `node` operation of the root.
3. Options for all children can be given between the root node and the first child.
4. Options applying to a specific child path are given as options to the `child` operation.
5. Options applying to the node of a child, but not to the whole child path, are given as options to the `node` command inside the ⟨*child path*⟩.

```
\begin{tikzpicture}
  \scoped
    [...]              % Options apply to the whole tree
    \node[...] {root}  % Options apply to the root node only
       [...]           % Options apply to all children
       child[...]      % Options apply to this child and all its children
       {
         node[...] {}  % Options apply to the child node only
         ...
       }
       child[...]      % Options apply to this child and all its children
    ;
\end{tikzpicture}
```

There are additional styles that influence how children are rendered:

/tikz/every child (style, initially empty)

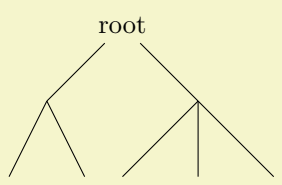
This style is used at the beginning of each child, as if you had given the style's contents as options to the `child` operation.

/tikz/every child node (style, initially empty)

This style is used at the beginning of each child node in addition to the `every node` style.

/tikz/level=⟨*number*⟩ (style, no default, initially empty)

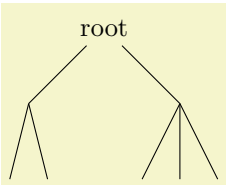
This style is executed at the beginning of each set of children, where ⟨*number*⟩ is the current level in the current tree. For example, when you say `\node {x} child child;`, then `level=1` is used before the first `child`. The style or code of this key will be passed ⟨*number*⟩ as its first parameter. If this first `child` has children itself, then `level=2` would be used for them.




```
\begin{tikzpicture}[level/.style={sibling distance=20mm/#1}]
  \node {root}
    child { child child }
    child { child child child };
\end{tikzpicture}
```

/tikz/level ⟨*number*⟩ (style, initially empty)

This style is used in addition to the `level` style. So, when you say `\node {x} child child;`, then the following key list is executed: `level=1,level 1`.




```
\begin{tikzpicture}
  [level 1/.style={sibling distance=20mm},
   level 2/.style={sibling distance=5mm}]
  \node {root}
    child { child child }
    child { child child child };
\end{tikzpicture}
```

21.5 Placing Child Nodes

21.5.1 Basic Idea

Perhaps the most difficult part in drawing a tree is the correct layout of the children. Typically, the children have different sizes and it is not easy to arrange them in such a manner that not too much space is wasted, the children do not overlap, and they are either evenly spaced or their centers are evenly distributed. Calculating good positions is especially difficult since a good position for the first child may depend on the size of the last child.

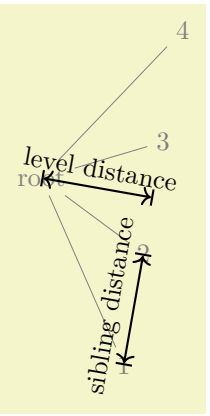
In basic Ti*k*Z, when you do not make use of the graph drawing facilities explained in Part IV, a comparatively simple approach is taken to placing the children. In order to compute a child's position, all that is taken into account is the number of the current child in the list of children and the number of children in this list. Thus, if a node has five children, then there is a fixed position for the first child, a position for the

second child, and so on. These positions *do not depend on the size of the children* and, hence, children can easily overlap. However, since you can use options to shift individual children a bit, this is not as great a problem as it may seem.

Although the placement of the children only depends on their number in the list of children and the total number of children, everything else about the placement is highly configurable. You can change the distance between children (appropriately called the `sibling distance`) and the distance between levels of the tree. These distances may change from level to level. The direction in which the tree grows can be changed globally and for parts of the tree. You can even specify your own "growth function" to arrange children on a circle or along special lines or curves.

21.5.2 Default Growth Function

The default growth function works as follows: Assume that we are given a node and five children. These children will be placed on a line with their centers (or, more generally, with their anchors) spaced apart by the current `sibling distance`. The line is orthogonal to the current *direction of growth*, which is set with the `grow` and `grow'` option (the latter option reverses the ordering of the children). The distance from the line to the parent node is given by the `level distance`.



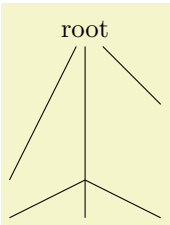

```
\begin{tikzpicture}[sibling distance=15mm, level distance=15mm]
  \path [help lines]
    node (root) {root}
    [grow=-10]
    child {node {1}}
    child {node {2}}
    child {node {3}}
    child {node {4}};

  \draw[|<->|,thick] (root-1.center)
    -- node[above,sloped] {sibling distance} (root-2.center);

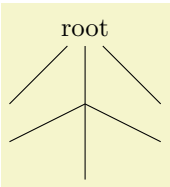
  \draw[|<->|,thick] (root.center)
    -- node[above,sloped] {level distance} +(-10:\tikzleveldistance);
\end{tikzpicture}
```

`/tikz/level distance`=⟨*distance*⟩ (no default, initially `15mm`)

This key determines the distance between different levels of the tree, more precisely, between the parent and the line on which its children are arranged. When given to a single child, this will set the distance for this child only.




```
\begin{tikzpicture}
  \node {root}
    [level distance=20mm]
    child
    child {
      [level distance=5mm]
      child
      child
      child
    }
    child[level distance=10mm];
\end{tikzpicture}
```



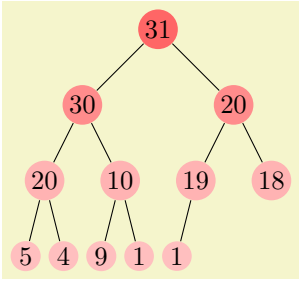

```
\begin{tikzpicture}
  [level 1/.style={level distance=10mm},
   level 2/.style={level distance=5mm}]
  \node {root}
    child
    child {
      child
      child[level distance=10mm]
      child
    }
    child;
\end{tikzpicture}
```

/tikz/sibling distance=⟨*distance*⟩ (no default, initially 15mm)

This key specifies the distance between the anchors of the children of a parent node.



```
\begin{tikzpicture}
  [level distance=4mm,
   level 1/.style={sibling distance=8mm},
   level 2/.style={sibling distance=4mm},
   level 3/.style={sibling distance=2mm}]
  \coordinate
     child {
       child {child child}
       child {child child}
     }
     child {
       child {child child}
       child {child child}
     };
\end{tikzpicture}
```




```
\begin{tikzpicture}
  [level distance=10mm,
   every node/.style={fill=red!60,circle,inner sep=1pt},
   level 1/.style={sibling distance=20mm,nodes={fill=red!45}},
   level 2/.style={sibling distance=10mm,nodes={fill=red!30}},
   level 3/.style={sibling distance=5mm,nodes={fill=red!25}}]
  \node {31}
     child {node {30}
       child {node {20}
         child {node {5}}
         child {node {4}}
       }
       child {node {10}
         child {node {9}}
         child {node {1}}
       }
     }
     child {node {20}
       child {node {19}
         child {node {1}}
         child[missing]
       }
       child {node {18}}
     };
\end{tikzpicture}
```

/tikz/grow=⟨*direction*⟩ (no default)

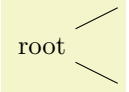
This key is used to define the ⟨*direction*⟩ in which the tree will grow. The ⟨*direction*⟩ can either be an angle in degrees or one of the following special text strings: `down`, `up`, `left`, `right`, `north`, `south`, `east`, `west`, `north east`, `north west`, `south east`, and `south west`. All of these have "their obvious meaning", so, say, `south west` is the same as the angle −135°.

As a side effect, this option installs the default growth function.

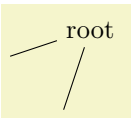
In addition to setting the direction, this option also has a seemingly strange effect: It sets the sibling distance for the current level to `0pt`, but leaves the sibling distance for later levels unchanged.

This somewhat strange behavior has a highly desirable effect: If you give this option before the list of children of a node starts, the "current level" is still the parent level. Each child will be on a later level and, hence, the sibling distance will be as specified originally. This will cause the children to be neatly aligned in a line orthogonal to the given ⟨*direction*⟩. However, if you give this option locally to a single child, then "current level" will be the same as the child's level. The zero sibling distance will then cause the child to be placed exactly at a point at distance `level distance` in the direction ⟨*direction*⟩. However, the children of the child will be placed "normally" on a line orthogonal to the ⟨*direction*⟩.

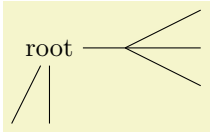
These placement effects are best demonstrated by some examples:



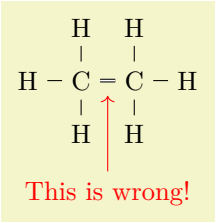

```
\tikz \node {root} [grow=right] child child;
```



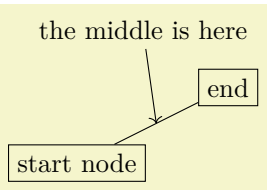

```
\tikz \node {root} [grow=south west] child child;
```




```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node {root}
    [grow=down]
    child
    child
    child[grow=right] {
      child child child
    };
\end{tikzpicture}
```




```
\begin{tikzpicture}[level distance=2em]
  \node {C}
    child[grow=up]    {node {H}}
    child[grow=left]  {node {H}}
    child[grow=down]  {node {H}}
    child[grow=right] {node {C}
        child[grow=up]    {node {H}}
        child[grow=right] {node {H}}
        child[grow=down]  {node {H}}
      edge from parent[double]
        coordinate (wrong)
    };
  \draw[<-,red] ([yshift=-2mm]wrong) -- +(0,-1)
    node[below]{This is wrong!};
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \node[rectangle,draw] (a) at (0,0) {start node};
  \node[rectangle,draw] (b) at (2,1) {end};

  \draw (a) -- (b)
    node[coordinate,midway] {}
      child[grow=100,<-] {node[above] {the middle is here}};
\end{tikzpicture}
```

`/tikz/grow'`=⟨*direction*⟩ (no default)

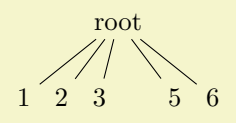
This key has the same effect as `grow`, only the children are arranged in the opposite order.

21.5.3 Missing Children

Sometimes one or more of the children of a node are "missing". Such a missing child will count as a child with respect to the total number of children and also with respect to the current child count, but it will not be rendered.

`/tikz/missing`=⟨*true or false*⟩ (default `true`)

If this option is given to a child, the current child counter is increased, but the child is otherwise ignored. In particular, the normal contents of the child is completely ignored.



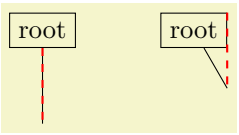

```
\begin{tikzpicture}[level distance=10mm,sibling distance=5mm]
  \node {root} [grow=down]
    child          { node {1} }
    child          { node {2} }
    child          { node {3} }
    child[missing] { node {4} }
    child          { node {5} }
    child          { node {6} };
\end{tikzpicture}
```

21.5.4 Custom Growth Functions

`/tikz/growth parent anchor=`⟨*anchor*⟩ (no default, initially `center`)

This key allows you to specify which anchor of the parent node is to be used for computing the children's position. For example, when there is only one child and the `level distance` is `2cm`, then the child node will be placed two centimeters below the ⟨*anchor*⟩ of the parent node. "Being placed" means that the child node's anchor (which is the anchor specified using the `anchor=` option in the `node` command of the child) is two centimeters below the parent node's ⟨*anchor*⟩.

In the following example, the two red lines both have length `1cm`.

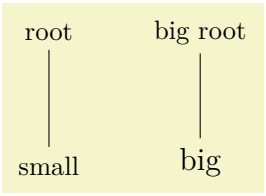



```
\begin{tikzpicture}[level distance=1cm]
  \node [rectangle,draw] (a) at (0,0) {root}
  [growth parent anchor=south] child;

  \node [rectangle,draw] (b) at (2,0) {root}
  [growth parent anchor=north east] child;

  \draw [red,thick,dashed] (a.south) -- (a-1);
  \draw [red,thick,dashed] (b.north east) -- (b-1);
\end{tikzpicture}
```

In the next example, the top and bottom nodes are aligned at the top and the bottom, respectively.




```
\begin{tikzpicture}
  [level distance=2cm,growth parent anchor=north,
   every node/.style={anchor=north,rectangle,draw}
   every child node/.style={anchor=south}]

  \node at (0,0) {root} child {node {small}};

  \node at (2,0) {big root} child {node {\large big}};
\end{tikzpicture}
```

`/tikz/growth function=`⟨*macro name*⟩ (no default, initially an internal function)

This rather low-level option allows you to set a new growth function. The ⟨*macro name*⟩ must be the name of a macro without parameters. This macro will be called for each child of a node. The initial function is an internal function that corresponds to downward growth.

The effect of executing the macro should be the following: It should transform the coordinate system in such a way that the origin becomes the place where the current child should be anchored. When the macro is called, the current coordinate system will be set up such that the anchor of the parent node is in the origin. Thus, in each call, the ⟨*macro name*⟩ must essentially do a shift to the child's origin. When the macro is called, the TEX counter `\tikznumberofchildren` will be set to the total number of children of the parent node and the counter `\tikznumberofcurrentchild` will be set to the number of the current child.

The macro may, in addition to shifting the coordinate system, also transform the coordinate system further. For example, it could be rotated or scaled.

Additional growth functions are defined in the library, see Section 76.

21.6 Edges From the Parent Node

Every child node is connected to its parent node via a special kind of edge called the `edge from parent`. This edge is added to the ⟨*child path*⟩ when the following path operation is encountered:

`\path` … `edge from parent[`⟨*options*⟩`]` … `;`

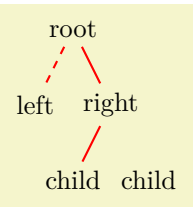
This path operation can only be used inside ⟨*child paths*⟩ and should be given at the end, possibly followed by ⟨*node specifications*⟩ like `node {a}`. If a ⟨*child path*⟩ does not contain this operation, it will be added at the end of the ⟨*child path*⟩ automatically.

By default, this operation does the following:

1. The following style is executed:

/tikz/edge from parent (style, initially draw)

This style is inserted right before the edge from parent path and before the ⟨*options*⟩ are inserted.




```
\begin{tikzpicture}
  [edge from parent/.style={draw,red,thick}]
  \node {root}
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

2. Next, the ⟨*options*⟩ are executed.
3. Next, the text stored in the following key is inserted:

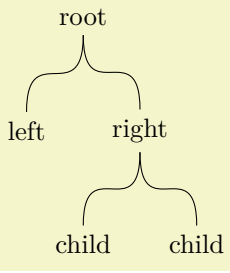
/tikz/edge from parent path=⟨*path*⟩ (no default, initially code shown below)

This option allows you to set the edge from parent path to a new path. Initially, this path is the following:

```
(\tikzparentnode\tikzparentanchor) -- (\tikzchildnode\tikzchildanchor)
```

The \tikzparentnode is a macro that will expand to the name of the parent node. This works even when you have not assigned a name to the parent node, in this case an internal name is automatically generated. The \tikzchildnode is a macro that expands to the name of the child node. The two ...anchor macros are empty by default. So, what is essentially inserted is just the path segment (\tikzparentnode) -- (\tikzchildnode); which is exactly an edge from the parent to the child.

You can modify this edge from parent path to achieve all sorts of effects. For example, we could replace the straight line by a curve as follows:




```
\begin{tikzpicture}[level distance=15mm, sibling distance=15mm,
  edge from parent path=
  {(\tikzparentnode.south) .. controls +(0,-1) and +(0,1)
                           .. (\tikzchildnode.north)}]
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

Further useful edge from parent paths are defined in the tree library, see Section 76.

The nodes in a ⟨*node specification*⟩ following the edge from parent path command get executed as if the pos option had been added to all these nodes, see also Section 17.8.

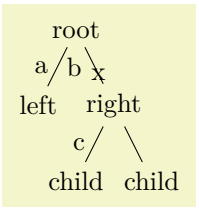
As an example, consider the following code:

```
\node (root) {} child {node (child) {} edge to parent node {label}};
```

The edge to parent operation and the following node operation will, together, have the same effect as if we had said:

```
(root) -- (child) node [pos=0.5] {label}
```

Here is a more complicated example:



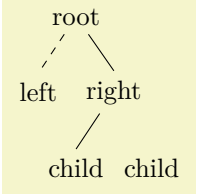

```
\begin{tikzpicture}
  \node {root}
    child {
      node {left}
      edge from parent
        node[left] {a}
        node[right] {b}
    }
    child {
      node {right}
        child {
          node {child}
          edge from parent
            node[left] {c}
        }
        child {node {child}}
      edge from parent
        node[near end] {x}
    };
\end{tikzpicture}
```

As said before, the anchors in the default `edge from parent path` are empty. However, you can set them using the following keys:

`/tikz/child anchor=`⟨*anchor*⟩ (no default, initially `border`)

Specifies the anchor where the edge from parent meets the child node by setting the macro `\tikzchildanchor` to `.`⟨*anchor*⟩.

If you specify `border` as the ⟨*anchor*⟩, then the macro `\tikzchildanchor` is set to the empty string. The effect of this is that the edge from the parent will meet the child on the border at an automatically calculated position.




```
\begin{tikzpicture}
  \node {root}
    [child anchor=north]
    child {node {left} edge from parent[dashed]}
    child {node {right}
      child {node {child}}
      child {node {child} edge from parent[draw=none]}
    };
\end{tikzpicture}
```

`/tikz/parent anchor=`⟨*anchor*⟩ (no default, initially `border`)

This option works the same way as the `child anchor`, only for the parent.

All of the above describes the standard functioning of the `edge from parent` command. You may, however, sometimes need even more fine-grained control (the graph drawing engine needs it, for instance). In such cases the following key gives you complete control:

`/tikz/edge from parent macro=`⟨*macro*⟩ (no default)

The ⟨*macro*⟩ gets expanded each time the `edge from parent` path operation is used. This ⟨*macro*⟩ must take two parameters and must expand to some text that is subsequently parsed by the parser. The first parameter will be the set of ⟨*options*⟩ that where passed to the `edge from parent` command, the second parameter will be the ⟨*node specifications*⟩ that following the command.

The standard behavior of drawing a straight line from the parent node to the child node could be achieved by setting the ⟨*macro*⟩ to the following:

```
\def\mymacro#1#2{
  [style=edge from parent, #1]
  (\tikzparentnode\tikzparentanchor) -- #2 (\tikzchildnode\tikzchildanchor)
}
```

Note that `#2` is placed between `--` and the node to ensure that nodes are put “on top” of the line.

22 Plots of Functions

A warning before we get started: *If you are looking for an easy way to create a normal plot of a function with scientific axes, ignore this section and instead look at the* `pgfplots` *package or at the* `datavisualization` *command from Part VI.*

22.1 Overview

Ti*k*Z can be used to create plots of functions, a job that is normally handled by powerful programs like GNUPLOT or MATHEMATICA. These programs can produce two different kinds of output: First, they can output a complete plot picture in a certain format (like PDF) that includes all low-level commands necessary for drawing the complete plot (including axes and labels). Second, they can usually also produce "just plain data" in the form of a long list of coordinates. Most of the powerful programs consider it a to be "a bit boring" to just output tabled data and very much prefer to produce fancy pictures. Nevertheless, when coaxed, they can also provide the plain data.

The advantage of creating plots directly using Ti*k*Z is *consistency:* Plots created using Ti*k*Z will automatically have the same styling and fonts as those used in the rest of a document – something that is hard to do right when an external program gets involved. Other problems people encounter with external programs include: Formulas will look different, if they can be rendered at all; line widths will usually be too thick or too thin; scaling effects upon inclusion can create a mismatch between sizes in the plot and sizes in the text; the automatic grid generated by most programs is mostly distracting; the automatic ticks generated by most programs are cryptic numerics (try adding a tick reading "π" at the right point); most programs make it very easy to create "chart junk" in a most convenient fashion; arrows and plot marks will almost never match the arrows used in the rest of the document. This list is not exhaustive, unfortunately.

There are basically three ways of creating plots using Ti*k*Z:

1. Use the `plot` path operation. How this works is explained in the present section. This is the most "basic" of the three options and forces you to do a lot of things "by hand" like adding axes or ticks.

2. Use the `datavisualization` path command, which is documented in Part VI. This command is much more powerful than the `plot` path operation and produces complete plots including axes and ticks. The downside is that you cannot use it to "just" quickly plot a simple curve (or, more precisely, it is hard to use it in this way).

3. Use the `pgfplots` package, which is basically an alternative to the `datavisualization` command. While the underlying philosophy of this package is not as "ambitious" as that of the command `datavisualization`, it is somewhat more mature, has a simpler design, and wider support base.

22.2 The Plot Path Operation

The `plot` path operation can be used to append a line or curve to the path that goes through a large number of coordinates. These coordinates are either given in a simple list of coordinates, read from some file, or they are computed on the fly.

The syntax of the `plot` comes in different versions.

`\path` … `--plot`⟨*further arguments*⟩ … ;

This operation plots the curve through the coordinates specified in the ⟨*further arguments*⟩. The current (sub)path is simply continued, that is, a line-to operation to the first point of the curve is implicitly added. The details of the ⟨*further arguments*⟩ will be explained in a moment.

`\path` … `plot`⟨*further arguments*⟩ … ;

This operation plots the curve through the coordinates specified in the ⟨*further arguments*⟩ by first "moving" to the first coordinate of the curve.

The ⟨*further arguments*⟩ are used in different ways to specifying the coordinates of the points to be plotted:

1. `--plot[`⟨*local options*⟩`]coordinates{`⟨*coordinate 1*⟩⟨*coordinate 2*⟩…⟨*coordinate n*⟩`}`

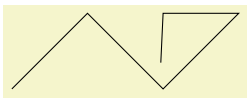
2. `--plot[`⟨*local options*⟩`]file{`⟨*filename*⟩`}`

3. `--plot[`⟨*local options*⟩`]`⟨*coordinate expression*⟩
4. `--plot[`⟨*local options*⟩`]function{`⟨*gnuplot formula*⟩`}`

These different ways are explained in the following.

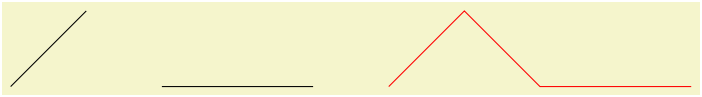
22.3 Plotting Points Given Inline

Points can be given directly in the TEX-file as in the following example:



```
\tikz \draw plot coordinates {(0,0) (1,1) (2,0) (3,1) (2,1) (10:2cm)};
```

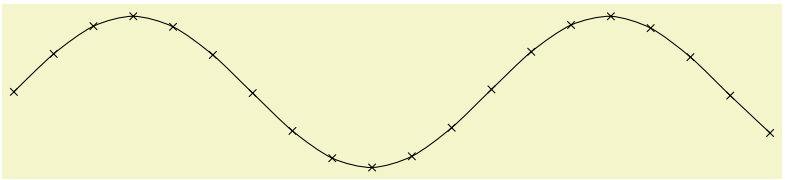
Here is an example showing the difference between `plot` and `--plot`:



```
\begin{tikzpicture}
  \draw (0,0) -- (1,1) plot coordinates {(2,0)  (4,0)};
  \draw[color=red,xshift=5cm]
        (0,0) -- (1,1) -- plot coordinates {(2,0)  (4,0)};
\end{tikzpicture}
```

22.4 Plotting Points Read From an External File

The second way of specifying points is to put them in an external file named ⟨*filename*⟩. Currently, the only file format that Ti*k*Z allows is the following: Each line of the ⟨*filename*⟩ should contain one line starting with two numbers, separated by a space. A line may also be empty or, if it starts with `#` or `%` it is considered empty. For such lines, a "new data set" is started, typically resulting in a new subpath being started in the plot (see Section 112.2.2 on how to change this behavior, if necessary). For lines containing two numbers, they must be separated by a space. They may be following by arbitrary text, which is ignored, *except* if it is `o` or `u`. In the first case, the point is considered to be an *outlier* and normally also results in a new subpath being started. In the second case, the point is considered to be *undefined*, which also results in a new subpath being started. Again, see Section 112.2.2 on how to change this, if necessary. (This is exactly the format that GNUPLOT produces when you say `set table`.)



```
\tikz \draw plot[mark=x,smooth] file {plots/pgfmanual-sine.table};
```

The file `plots/pgfmanual-sine.table` reads:

```
#Curve 0, 20 points
#x y type
0.00000 0.00000  i
0.52632 0.50235  i
1.05263 0.86873  i
1.57895 0.99997  i
...
9.47368 -0.04889  i
10.00000 -0.54402  i
```

It was produced from the following source, using `gnuplot`:

```
set table  "../plots/pgfmanual-sine.table"
set format "%.5f"
set samples 20
plot [x=0:10] sin(x)
```

The ⟨*local options*⟩ of the `plot` operation are local to each plot and do not affect other plots "on the same path". For example, `plot[yshift=1cm]` will locally shift the plot 1cm upward. Remember, however, that most options can only be applied to paths as a whole. For example, `plot[red]` does not have the effect of making the plot red. After all, you are trying to "locally" make part of the path red, which is not possible.

22.5 Plotting a Function

When you plot a function, the coordinates of the plot data can be computed by evaluating a mathematical expression. Since PGF comes with a mathematical engine, you can specify this expression and then have Ti*k*Z produce the desired coordinates for you, automatically.

Since this case is quite common when plotting a function, the syntax is easy: Following the `plot` command and its local options, you directly provide a ⟨*coordinate expression*⟩. It looks like a normal coordinate, but inside you may use a special macro, which is `\x` by default, but this can be changed using the `variable` option. The ⟨*coordinate expression*⟩ is then evaluated for different values for `\x` and the resulting coordinates are plotted.

Note that you will often have to put the x- or y-coordinate inside braces, namely whenever you use an expression involving a parenthesis.

The following options influence how the ⟨*coordinate expression*⟩ is evaluated:

`/tikz/variable`=⟨*macro*⟩ (no default, initially `\x`)

Sets the macro whose value is set to the different values when ⟨*coordinate expression*⟩ is evaluated.

`/tikz/samples`=⟨*number*⟩ (no default, initially `25`)

Sets the number of samples used in the plot.

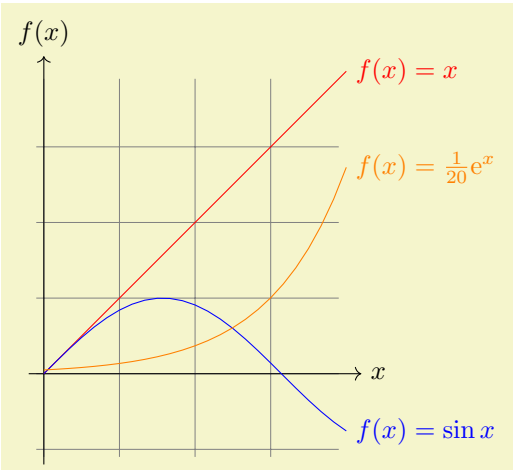
`/tikz/domain`=⟨*start*⟩`:`⟨*end*⟩ (no default, initially `-5:5`)

Sets the domain from which the samples are taken.

`/tikz/samples at`=⟨*sample list*⟩ (no default)

This option specifies a list of positions for which the variable should be evaluated. For instance, you can say `samples at={1,2,8,9,10}` to have the variable evaluated exactly for values 1, 2, 8, 9, and 10. You can use the `\foreach` syntax, so you can use `...` inside the ⟨*sample list*⟩.

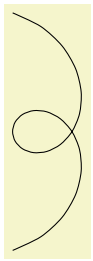
When this option is used, the `samples` and `domain` option are overruled. The other way round, setting either `samples` or `domain` will overrule this option.



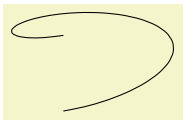

```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[color=red]    plot (\x,\x)             node[right] {$f(x) =x$};
  % \x r means to convert '\x' from degrees to _r_adians:
  \draw[color=blue]   plot (\x,{sin(\x r)})    node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot (\x,{0.05*exp(\x)}) node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth,variable=\t]
  plot ({\t*sin(\t r)},{\t*cos(\t r)});
```



```
\tikz \draw[domain=0:360,smooth,variable=\t]
  plot ({sin(\t)},\t/360,{cos(\t)});
```

22.6 Plotting a Function Using Gnuplot

Often, you will want to plot points that are given via a function like $f(x) = x \sin x$. Unfortunately, TEX does not really have enough computational power to generate the points of such a function efficiently (it is a text processing program, after all). However, if you allow it, TEX can try to call external programs that can easily produce the necessary points. Currently, Ti*k*Z knows how to call GNUPLOT.

When Ti*k*Z encounters your operation `plot[id=`⟨*id*⟩`] function{x*sin(x)}` for the first time, it will create a file called ⟨*prefix*⟩⟨*id*⟩`.gnuplot`, where ⟨*prefix*⟩ is `\jobname.` by default, that is, the name of your main `.tex` file. If no ⟨*id*⟩ is given, it will be empty, which is alright, but it is better when each plot has a unique ⟨*id*⟩ for reasons explained in a moment. Next, Ti*k*Z writes some initialization code into this file followed by `plot x*sin(x)`. The initialization code sets up things such that the `plot` operation will write the coordinates into another file called ⟨*prefix*⟩⟨*id*⟩`.table`. Finally, this table file is read as if you had said `plot file{`⟨*prefix*⟩⟨*id*⟩`.table}`.

For the plotting mechanism to work, two conditions must be met:

1. You must have allowed TEX to call external programs. This is often switched off by default since this is a security risk (you might, without knowing, run a TEX file that calls all sorts of "bad" commands). To enable this "calling external programs" a command line option must be given to the TEX program. Usually, it is called something like `shell-escape` or `enable-write18`. For example, for my `pdflatex` the option `--shell-escape` can be given.

2. You must have installed the `gnuplot` program and TEX must find it when compiling your file.

Unfortunately, these conditions will not always be met. Especially if you pass some source to a coauthor and the coauthor does not have GNUPLOT installed, he or she will have trouble compiling your files.

For this reason, Ti*k*Z behaves differently when you compile your graphic for the second time: If upon reaching `plot[id=`⟨*id*⟩`] function{...}` the file ⟨*prefix*⟩⟨*id*⟩`.table` already exists *and* if the ⟨*prefix*⟩⟨*id*⟩`.gnuplot` file contains what Ti*k*Z thinks that it "should" contain, the `.table` file is immediately read without trying to call a `gnuplot` program. This approach has the following advantages:

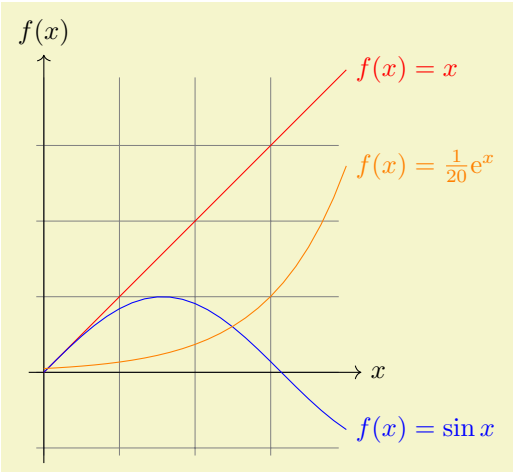
1. If you pass a bundle of your `.tex` file and all `.gnuplot` and `.table` files to someone else, that person can TEX the `.tex` file without having to have `gnuplot` installed.

2. If the `\write18` feature is switched off for security reasons (a good idea), then, upon the first compilation of the `.tex` file, the `.gnuplot` will still be generated, but not the `.table` file. You can then simply call `gnuplot` "by hand" for each `.gnuplot` file, which will produce all necessary `.table` files.

3. If you change the function that you wish to plot or its domain, Ti*k*Z will automatically try to regenerate the `.table` file.

4. If, out of laziness, you do not provide an `id`, the same `.gnuplot` will be used for different plots, but this is not a problem since the `.table` will automatically be regenerated for each plot on-the-fly. *Note: If you intend to share your files with someone else, always use an id, so that the file can by typeset without having* GNUPLOT *installed.* Also, having unique ids for each plot will improve compilation speed since no external programs need to be called, unless it is really necessary.

When you use `plot function{`⟨*gnuplot formula*⟩`}`, the ⟨*gnuplot formula*⟩ must be given in the `gnuplot` syntax, whose details are beyond the scope of this manual. Here is the ultra-condensed essence: Use `x` as the variable and use the C-syntax for normal plots, use `t` as the variable for parametric plots. Here are some examples:




```
\begin{tikzpicture}[domain=0:4]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

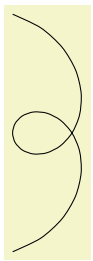
  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[color=red]    plot[id=x]   function{x}            node[right] {$f(x) =x$};
  \draw[color=blue]   plot[id=sin] function{sin(x)}       node[right] {$f(x) = \sin x$};
  \draw[color=orange] plot[id=exp] function{0.05*exp(x)} node[right] {$f(x) = \frac{1}{20} \mathrm e^x$};
\end{tikzpicture}
```

The plot is influenced by the following options: First, the options `samples` and `domain` explained earlier. Second, there are some more specialized options.

`/tikz/parametric=`⟨*boolean*⟩ (default `true`)

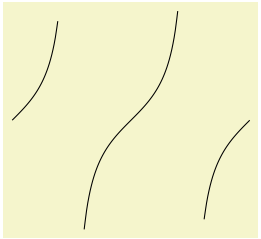
Sets whether the plot is a parametric plot. If true, then `t` must be used instead of `x` as the parameter and two comma-separated functions must be given in the ⟨*gnuplot formula*⟩. An example is the following:



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth]
  plot[parametric,id=parametric-example] function{t*sin(t),t*cos(t)};
```

`/tikz/range=`⟨*start*⟩`:`⟨*end*⟩ (no default)

This key sets the range of the plot. If set, all points whose y-coordinates lie outside this range will be considered to be outliers and will cause jumps in the plot, by default:



```
\tikz \draw[scale=0.5,domain=-3.141:3.141, samples=100, smooth, range=-3:3]
  plot[id=tan-example] function{tan(x)};
```

/tikz/yrange=⟨*start*⟩:⟨*end*⟩ (no default)

Same as `range`.

/tikz/xrange=⟨*start*⟩:⟨*end*⟩ (no default)

Set the x-range. This makes sense only for parametric plots.



```
\tikz \draw[scale=0.5,domain=-3.141:3.141,smooth,xrange=0:1]
  plot[parametric,id=parametric-example-cut] function{t*sin(t),t*cos(t)};
```

/tikz/id=⟨*id*⟩ (no default)

Sets the identifier of the current plot. This should be a unique identifier for each plot (though things will also work if it is not, but not as well, see the explanations above). The ⟨*id*⟩ will be part of a filename, so it should not contain anything fancy like `*` or `$`.

/tikz/prefix=⟨*prefix*⟩ (no default)

The ⟨*prefix*⟩ is put before each plot file name. The default is `\jobname.`, but if you have many plots, it might be better to use, say `plots/` and have all plots placed in a directory. You have to create the directory yourself.

/tikz/raw gnuplot (no value)

This key causes the ⟨*gnuplot formula*⟩ to be passed on to GNUPLOT without setting up the samples or the `plot` operation. Thus, you could write

```
plot[raw gnuplot,id=raw-example] function{set samples 25; plot sin(x)}
```

This can be useful for complicated things that need to be passed to GNUPLOT. However, for really complicated situations you should create a special external generating GNUPLOT file and use the `file`-syntax to include the table "by hand".

The following styles influence the plot:

/tikz/every plot (style, initially empty)

This style is installed in each plot, that is, as if you always said

```
  plot[every plot,...]
```

This is most useful for globally setting a prefix for all plots by saying:

```
\tikzset{every plot/.style={prefix=plots/}}
```

22.7 Placing Marks on the Plot

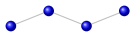
As we saw already, it is possible to add *marks* to a plot using the `mark` option. When this option is used, a copy of the plot mark is placed on each point of the plot. Note that the marks are placed *after* the whole path has been drawn/filled/shaded. In this respect, they are handled like text nodes.

In detail, the following options govern how marks are drawn:

/tikz/mark=⟨*mark mnemonic*⟩ (no default)

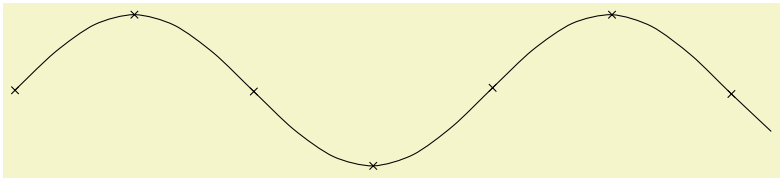
Sets the mark to a mnemonic that has previously been defined using the `\pgfdeclareplotmark`. By default, `*`, `+`, and `x` are available, which draw a filled circle, a plus, and a cross as marks. Many more marks become available when the library `plotmarks` is loaded. Section 65.6 lists the available plot marks.

One plot mark is special: the `ball` plot mark is available only in Ti*k*Z. The `ball color` option determines the balls's color. Do not use this option with a large number of marks since it will take very long to render in PostScript.

Option	Effect
`mark=ball`	

`/tikz/mark repeat=`⟨*r*⟩ (no default)

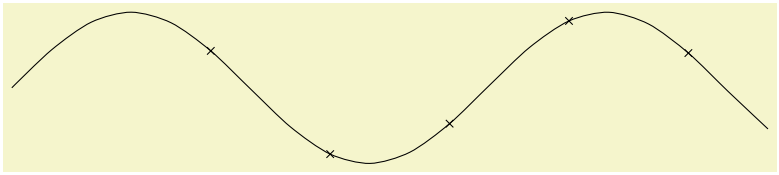
This option tells Ti*k*Z that only every *r*th mark should be drawn.



```
\tikz \draw plot[mark=x,mark repeat=3,smooth] file {plots/pgfmanual-sine.table};
```

`/tikz/mark phase=`⟨*p*⟩ (no default)

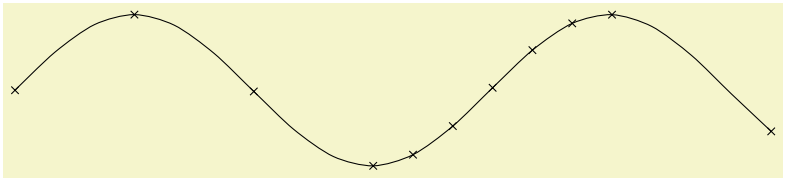
This option tells Ti*k*Z that the first mark to be draw should be the *p*th, followed by the $(p+r)$th, then the $(p+2r)$th, and so on.



```
\tikz \draw plot[mark=x,mark repeat=3,mark phase=6,smooth] file {plots/pgfmanual-sine.table};
```

`/tikz/mark indices=`⟨*list*⟩ (no default)

This option allows you to specify explicitly the indices at which a mark should be placed. Counting starts with 1. You can use the `\foreach` syntax, that is, `...` can be used.



```
\tikz \draw plot[mark=x,mark indices={1,4,...,10,11,12,...,16,20},smooth]
  file {plots/pgfmanual-sine.table};
```

`/tikz/mark size=`⟨*dimension*⟩ (no default)

Sets the size of the plot marks. For circular plot marks, ⟨*dimension*⟩ is the radius, for other plot marks ⟨*dimension*⟩ should be about half the width and height.

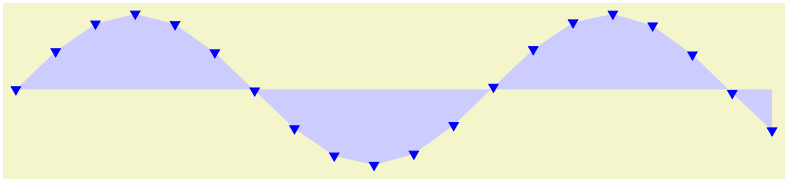
This option is not really necessary, since you achieve the same effect by specifying `scale=`⟨*factor*⟩ as a local option, where ⟨*factor*⟩ is the quotient of the desired size and the default size. However, using `mark size` is a bit faster and more natural.

`/tikz/every mark` (style, no value)

This style is installed before drawing plot marks. For example, you can scale (or otherwise transform) the plot mark or set its color.

`/tikz/mark options=`⟨*options*⟩ (no default)

Redefines `every mark` such that it sets `{`⟨*options*⟩`}`.



```
\tikz \fill[fill=blue!20]
  plot[mark=triangle*,mark options={color=blue,rotate=180}]
    file{plots/pgfmanual-sine.table} |- (0,0);
```

`/tikz/no marks` (style, no value)

Disables markers (the same as `mark=none`).

`/tikz/no markers` (style, no value)

Disables markers (the same as `mark=none`).

22.8 Smooth Plots, Sharp Plots, Jump Plots, Comb Plots and Bar Plots

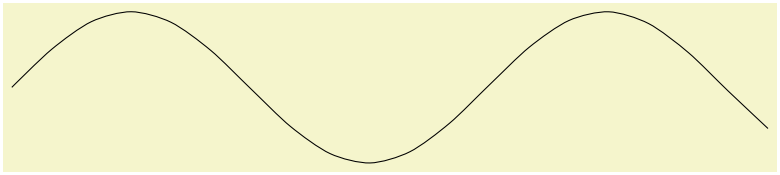
There are different things the `plot` operation can do with the points it reads from a file or from the inlined list of points. By default, it will connect these points by straight lines. However, you can also use options to change the behavior of `plot`.

`/tikz/sharp plot` (no value)

This is the default and causes the points to be connected by straight lines. This option is included only so that you can "switch back" if you "globally" install, say, `smooth`.

`/tikz/smooth` (no value)

This option causes the points on the path to be connected using a smooth curve:

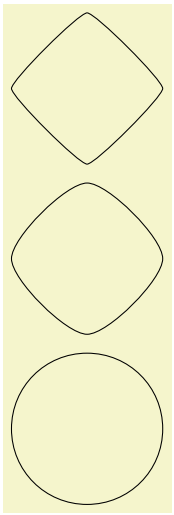


```
\tikz\draw plot[smooth] file{plots/pgfmanual-sine.table};
```

Note that the smoothing algorithm is not very intelligent. You will get the best results if the bending angles are small, that is, less than about 30° and, even more importantly, if the distances between points are about the same all over the plotting path.

`/tikz/tension=`⟨*value*⟩ (no default)

This option influences how "tight" the smoothing is. A lower value will result in sharper corners, a higher value in more "round" curves. A value of 1 results in a circle if four points at quarter-positions on a circle are given. The default is 0.55. The "correct" value depends on the details of plot.



```
\begin{tikzpicture}[smooth cycle]
  \draw                  plot[tension=0.2]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-2.25cm] plot[tension=0.5]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
  \draw[yshift=-4.5cm]  plot[tension=1]
    coordinates{(0,0) (1,1) (2,0) (1,-1)};
\end{tikzpicture}
```

/tikz/smooth cycle (no value)

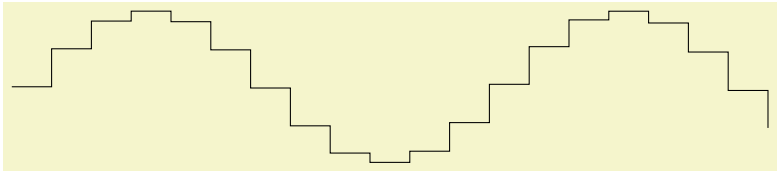
This option causes the points on the path to be connected using a closed smooth curve.



```
\tikz[scale=0.5]
  \draw plot[smooth cycle] coordinates{(0,0) (1,0) (2,1) (1,2)}
       plot                coordinates{(0,0) (1,0) (2,1) (1,2)} -- cycle;
```

/tikz/const plot (no value)

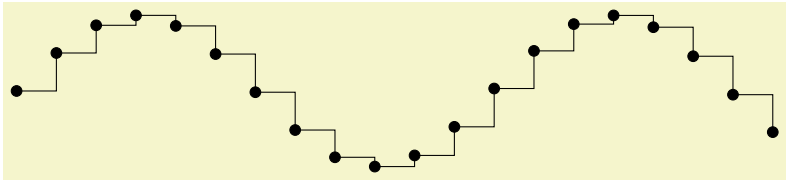
This option causes the points on the path to be connected using piecewise constant series of lines:



```
\tikz\draw plot[const plot] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark left (no value)

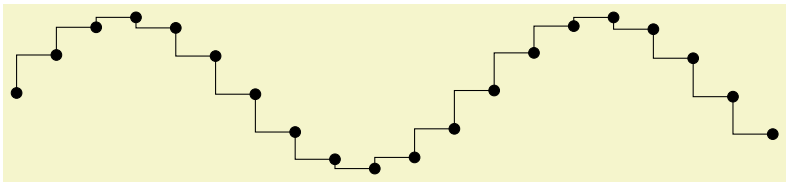
Just an alias for /tikz/const plot.



```
\tikz\draw plot[const plot mark left,mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark right (no value)

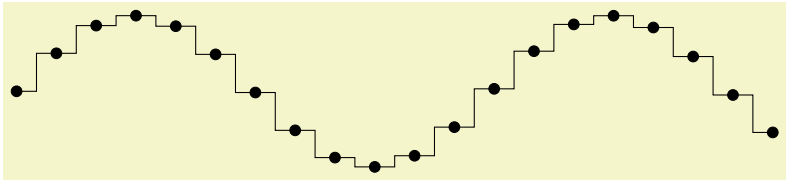
A variant of /tikz/const plot which places its mark on the right ends:



```
\tikz\draw plot[const plot mark right,mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/const plot mark mid (no value)

A variant of `/tikz/const plot` which places its mark in the middle of the horizontal lines:

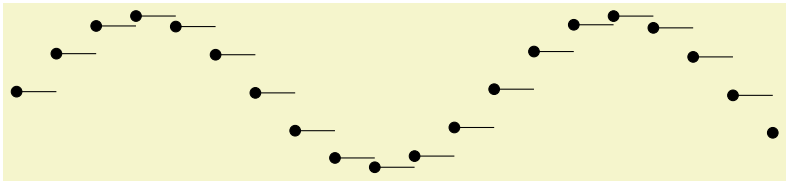


```
\tikz\draw plot[const plot mark mid,mark=*] file{plots/pgfmanual-sine.table};
```

More precisely, it generates vertical lines in the middle between each pair of consecutive points. If the mesh width is constant, this leads to symmetrically placed marks (“middle”).

/tikz/jump mark left (no value)

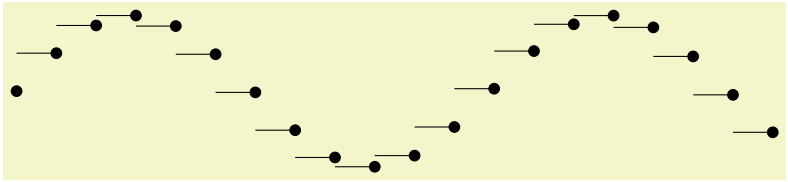
This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on left open ends:



```
\tikz\draw plot[jump mark left, mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/jump mark right (no value)

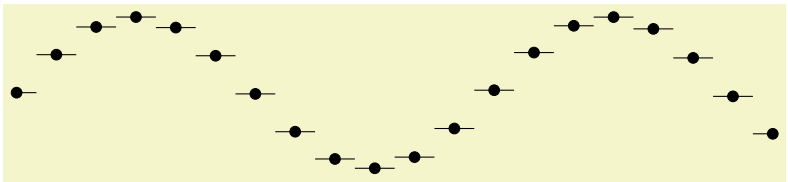
This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed on right open ends:



```
\tikz\draw plot[jump mark right, mark=*] file{plots/pgfmanual-sine.table};
```

/tikz/jump mark mid (no value)

This option causes the points on the path to be drawn using piecewise constant, non-connected series of lines. If there are any marks, they will be placed in the middle of the horizontal line segments:

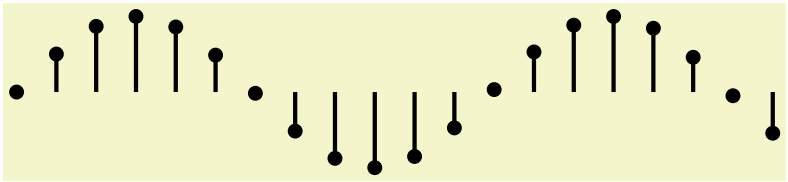


```
\tikz\draw plot[jump mark mid, mark=*] file{plots/pgfmanual-sine.table};
```

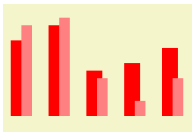
In case of non-constant mesh widths, the same remarks as for `const plot mark mid` apply.

/tikz/ycomb (no value)

This option causes the `plot` operation to interpret the plotting points differently. Instead of connecting them, for each point of the plot a straight line is added to the path from the x-axis to the point, resulting in a sort of "comb" or "bar diagram".



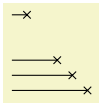
```
\tikz\draw[ultra thick] plot[ycomb,thin,mark=*] file{plots/pgfmanual-sine.table};
```



```
\begin{tikzpicture}[ycomb]
  \draw[color=red,line width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,line width=4pt,xshift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

/tikz/xcomb (no value)

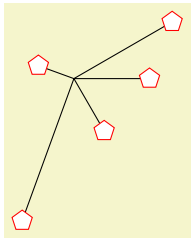
This option works like `ycomb` except that the bars are horizontal.



```
\tikz \draw plot[xcomb,mark=x] coordinates{(1,0) (0.8,0.2) (0.6,0.4) (0.2,1)};
```

/tikz/polar comb (no value)

This option causes a line from the origin to the point to be added to the path for each plot point.



```
\tikz \draw plot[polar comb,
    mark=pentagon*,mark options={fill=white,draw=red},mark size=4pt]
  coordinates {(0:1cm) (30:1.5cm) (160:.5cm) (250:2cm) (-60:.8cm)};
```

/tikz/ybar (no value)

This option produces fillable bar plots. It is thus very similar to `ycomb`, but it employs rectangular shapes instead of line-to operations. It thus allows to use any fill or pattern style.



```
\tikz\draw[draw=blue,fill=blue!60!black] plot[ybar] file{plots/pgfmanual-sine.table};
```

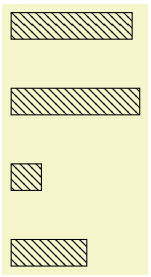


```
\begin{tikzpicture}[ybar]
  \draw[color=red,fill=red!80,bar width=6pt]
    plot coordinates{(0,1) (.5,1.2) (1,.6) (1.5,.7) (2,.9)};
  \draw[color=red!50,fill=red!20,bar width=4pt,bar shift=3pt]
    plot coordinates{(0,1.2) (.5,1.3) (1,.5) (1.5,.2) (2,.5)};
\end{tikzpicture}
```

The use of `bar width` and `bar shift` is explained in the `plothandlers` library documentation, section 65.4. Please refer to page 753.

`/tikz/xbar` (no value)

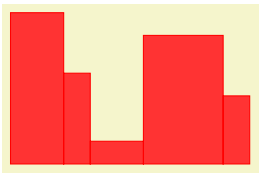
This option works like `ybar` except that the bars are horizontal.



```
\usetikzlibrary {patterns}
\tikz \draw[pattern=north west lines] plot[xbar]
   coordinates{(1,0) (0.4,1) (1.7,2) (1.6,3)};
```

`/tikz/ybar interval` (no value)

As `/tikz/ybar`, this options produces vertical bars. However, bars are centered at coordinate *intervals* instead of interval edges, and the bar's width is also determined relatively to the interval's length:



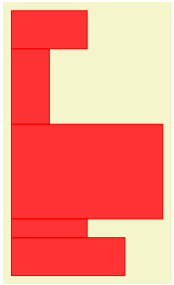
```
\begin{tikzpicture}[ybar interval,x=10pt]
  \draw[color=red,fill=red!80]
    plot coordinates{(0,2) (2,1.2) (3,.3) (5,1.7) (8,.9) (9,.9)};
\end{tikzpicture}
```

Since there are N intervals $[x_i, x_{i+1}]$ for given $N+1$ coordinates, you will always have one coordinate more than bars. The last y value will be ignored.

You can configure relative shifts and relative bar widths, which is explained in the `plothandlers` library documentation, section 65.4. Please refer to page 754.

`/tikz/xbar interval` (no value)

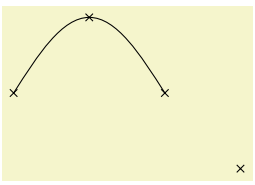
Works like `ybar interval`, but for horizontal bar plots.



```
\begin{tikzpicture}[xbar interval,x=0.5cm,y=0.5cm]
  \draw[color=red,fill=red!80]
    plot coordinates {(3,0) (2,1) (4,1.5) (1,4) (2,6) (2,7)};
\end{tikzpicture}
```

`/tikz/only marks` (no value)

This option causes only marks to be shown; no path segments are added to the actual path. This can be useful for quickly adding some marks to a path.



```
\tikz \draw (0,0) sin (1,1) cos (2,0)
  plot[only marks,mark=x] coordinates{(0,0) (1,1) (2,0) (3,-1)};
```

23 Transparency

23.1 Overview

Normally, when you paint something using any of Ti*k*Z's commands (this includes stroking, filling, shading, patterns, and images), the newly painted objects totally obscure whatever was painted earlier in the same area.

You can change this behavior by using something that can be thought of as "(semi)transparent colors". Such colors do not completely obscure the background, rather they blend the background with the new color. At first sight, using such semitransparent colors might seem quite straightforward, but the math going on in the background is quite involved and the correct handling of transparency fills some 64 pages in the PDF specification.

In the present section, we start with the different ways of specifying "how transparent" newly drawn objects should be. The simplest way is to just specify a percentage like "60% transparent". A much more general way is to use something that I call a *fading*, also known as a soft mask or a mask.

At the end of the section we address the problem of creating so-called *transparency groups*. This problem arises when you paint over a position several times with a semitransparent color. Sometimes you want the effect to accumulate, sometimes you do not.

Note: Transparency (or Opacity, as it may be called as well) is best supported by the pdfTEX driver. The SVG driver also has some support. The PostScript file format does not know about transparency. In `dvips`-generated PostScript files, transparency of graphic objects is defined through special commands that need further processing to become visible in the PDF output. For this, a recent version of Ghostscript, preferably 9.52 or newer, is required and its command line utility `ps2pdf` must be called with option `-dALLOWPSTRANSPARENCY`. Older versions may need option `-dNOSAFER` instead, but some advanced features, such as *transparency groups* and *fadings*, may not work at all. Printers and other programs will typically ignore opacity settings in PostScript files.

23.2 Specifying a Uniform Opacity

Specifying a stroke and/or fill opacity is quite easy using the following options.

`/tikz/draw opacity`=⟨*value*⟩ (no default)

This option sets "how transparent" lines should be. A value of `1` means "fully opaque" or "not transparent at all", a value of `0` means "fully transparent" or "invisible". A value of `0.5` yields lines that are semitransparent.

Note that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.



```
\begin{tikzpicture}[line width=1ex]
  \draw (0,0) -- (3,1);
  \filldraw [fill=yellow!80!black,draw opacity=0.5] (1,0) rectangle (2,1);
\end{tikzpicture}
```

Note that the `draw opacity` options only sets the opacity of drawn lines. The opacity of fillings is set using the option `fill opacity` (documented in Section 15.5.3. The option `opacity` sets both at the same time.

`/tikz/opacity`=⟨*value*⟩ (no default)

Sets both the drawing and filling opacity to ⟨*value*⟩.

The following predefined styles make it easier to use this option:

`/tikz/transparent` (style, no value)

Makes everything totally transparent and, hence, invisible.



```
\tikz{\fill[red]             (0,0)   rectangle (1,0.5);
      \fill[transparent,red] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/ultra nearly transparent (style, no value)

Makes everything, well, ultra nearly transparent.



```
\tikz{\fill[red]                    (0,0)   rectangle (1,0.5);
      \fill[ultra nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/very nearly transparent (style, no value)



```
\tikz{\fill[red]                    (0,0)   rectangle (1,0.5);
      \fill[very nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/nearly transparent (style, no value)



```
\tikz{\fill[red]                (0,0)   rectangle (1,0.5);
      \fill[nearly transparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/semitransparent (style, no value)



```
\tikz{\fill[red]             (0,0)   rectangle (1,0.5);
      \fill[semitransparent] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/nearly opaque (style, no value)



```
\tikz{\fill[red]          (0,0)   rectangle (1,0.5);
      \fill[nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/very nearly opaque (style, no value)



```
\tikz{\fill[red]               (0,0)   rectangle (1,0.5);
      \fill[very nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/ultra nearly opaque (style, no value)



```
\tikz{\fill[red]                (0,0)   rectangle (1,0.5);
      \fill[ultra nearly opaque] (0.5,0) rectangle (1.5,0.25); }
```

/tikz/opaque (style, no value)

This yields completely opaque drawings, which is the default.

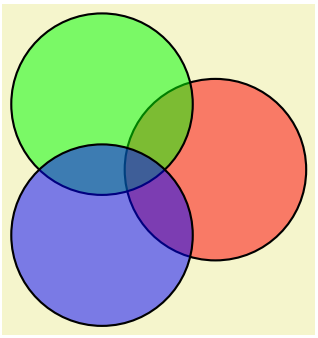


```
\tikz{\fill[red]    (0,0)   rectangle (1,0.5);
      \fill[opaque] (0.5,0) rectangle (1.5,0.25); }
```

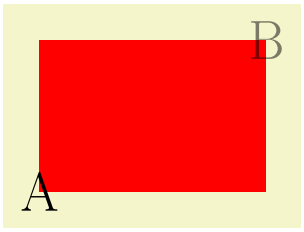
/tikz/fill opacity=⟨*value*⟩ (no default)

This option sets the opacity of fillings. In addition to filling operations, this opacity also applies to text and images.

Note, again, that when you use PostScript as your output format, this option works only with recent versions of Ghostscript.



```
\begin{tikzpicture}[thick,fill opacity=0.5]
  \filldraw[fill=red]   (0:1cm)    circle [radius=12mm];
  \filldraw[fill=green] (120:1cm)  circle [radius=12mm];
  \filldraw[fill=blue]  (-120:1cm) circle [radius=12mm];
\end{tikzpicture}
```

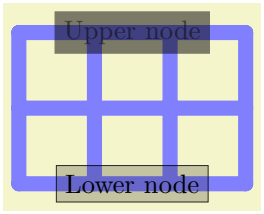



```
\begin{tikzpicture}
  \fill[red] (0,0) rectangle (3,2);

  \node                   at (0,0) {\huge A};
  \node[fill opacity=0.5] at (3,2) {\huge B};
\end{tikzpicture}
```

/tikz/text opacity=⟨*value*⟩ (no default)

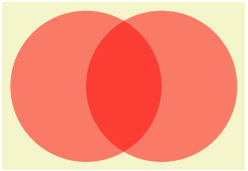
Sets the opacity of text labels, overriding the `fill opacity` setting.




```
\begin{tikzpicture}[every node/.style={fill,draw}]
  \draw[line width=2mm,blue!50,line cap=round] (0,0) grid (3,2);

  \node[opacity=0.5] at (1.5,2) {Upper node};
  \node[draw opacity=0.8,fill opacity=0.2,text opacity=1]
    at (1.5,0) {Lower node};
\end{tikzpicture}
```

Note the following effect: If you set up a certain opacity for stroking or filling and you stroke or fill the same area twice, the effect accumulates:



```
\begin{tikzpicture}[fill opacity=0.5]
  \fill[red] (0,0) circle [radius=1];
  \fill[red] (1,0) circle [radius=1];
\end{tikzpicture}
```

Often, this is exactly what you intend, but not always. You can use transparency groups, see the end of this section, to change this.

23.3 Blend Modes

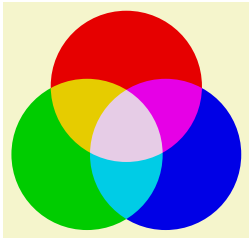
A *blend mode* specifies how colors mix when you paint on a canvas. Normally, if you paint a red box on a green circle, the red color will completely replace the green circle. However, in some situations you might also wish the red color to somehow "mix" or "blend" with the green circle. We already saw that, using transparency, we can draw something without completely obscuring the background. *Blending* is a similar operation, only here we mix colors in more complicated ways.

Note: Blending is a rather "advanced" feature of PDF. Most renderers, let alone printers, will have trouble rendering blending correctly.

/tikz/blend mode=⟨*mode*⟩ (no default)

Sets the current blend mode to ⟨*mode*⟩. Here ⟨*mode*⟩ must be one of the modes listed below. More details on these modes can also be found in Section 7.2.4 of the PDF Specification, version 1.7.

In the following example, the blend mode is only used and set inside a transparency group (see also Section 23.5). This is because most renderers (viewing programs) have trouble rendering blending correctly otherwise. For instance, at the time of writing, the versions of Adobe's Reader and Apple's Preview render the following drawing very differently, if the transparency group is not used in the following example.

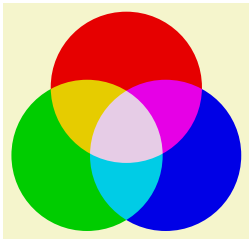


```
\tikz {
  \begin{scope}[transparency group]
    \begin{scope}[blend mode=screen]
      \fill[red!90!black]   ( 90:.6) circle [radius=1];
      \fill[green!80!black] (210:.6) circle [radius=1];
      \fill[blue!90!black]  (330:.6) circle [radius=1];
    \end{scope}
  \end{scope}
}
```

Because of the trouble with rendering blending correctly outside transparency groups, there is a special key that establishes a transparency group and sets a blend mode simultaneously:

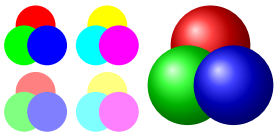
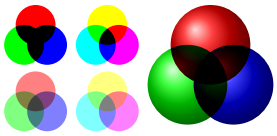
`/tikz/blend group=`⟨*mode*⟩ (no default)

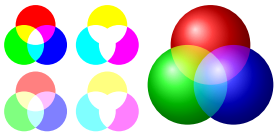
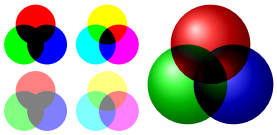
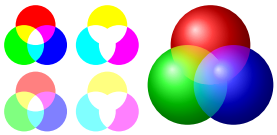
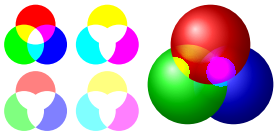
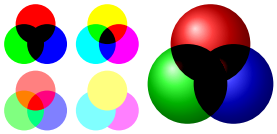
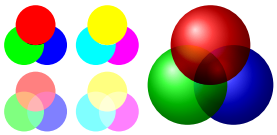
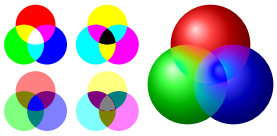
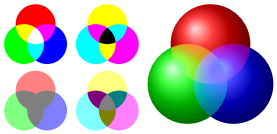
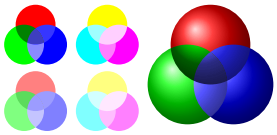
This key can only be used with a scope (like `transparency group`). It will cause the current scope to become a transparency group and, inside this group, the blend mode will be set to ⟨*mode*⟩.

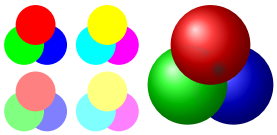
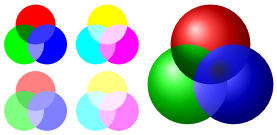
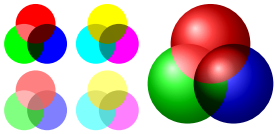


```
\tikz [blend group=screen] {
  \fill[red!90!black]   ( 90:.6) circle [radius=1];
  \fill[green!80!black] (210:.6) circle [radius=1];
  \fill[blue!90!black]  (330:.6) circle [radius=1];
}
```

Here is an overview of the effects of the different available blend modes. In the examples, we always have three circles drawn on top of each other (as in the example code earlier): We start with a triple of pure red, green, and blue. Below it, we have a triple of light versions of these three colors (`red!50`, `green!50`, and `blue!50`). Next comes the triple yellow, cyan, and magenta; again with a triple of light versions below it. The large example consists of three balls (produced using `ball color`) having the colors red, green, and blue, are drawn on top of each other just like the circles.

Example	*Mode*	*Explanations quoted from Table 7.2 of the PDF Specification, Version 1.7*
	`normal`	When painting a pixel with a some color (called the "source color"), the background color (called the "backdrop") is completely ignored.
	`multiply`	Multiplies the backdrop and source color values. The result color is always at least as dark as either of the two constituent colors. Multiplying any color with black produces black; multiplying with white leaves the original color unchanged. Painting successive overlapping objects with a color other than black or white produces progressively darker colors.

	`screen`	Multiplies the complements of the backdrop and source color values, then complements the result. The result color is always at least as light as either of the two constituent colors. Screening any color with white produces white; screening with black leaves the original color unchanged. The effect is similar to projecting multiple photographic slides simultaneously onto a single screen.
	`overlay`	Multiplies or screens the colors, depending on the backdrop color value. Source colors overlay the backdrop while preserving its highlights and shadows. The backdrop color is not replaced but is mixed with the source color to reflect the lightness or darkness of the backdrop.
	`darken`	Selects the darker of the backdrop and source colors. The backdrop is replaced with the source where the source is darker; otherwise, it is left unchanged.
	`lighten`	Selects the lighter of the backdrop and source colors. The backdrop is replaced with the source where the source is lighter; otherwise, it is left unchanged.
	`color dodge`	Brightens the backdrop color to reflect the source color. Painting with black produces no changes.
	`color burn`	Darkens the backdrop color to reflect the source color. Painting with white produces no change.
	`hard light`	Multiplies or screens the colors, depending on the source color value. The effect is similar to shining a harsh spotlight on the backdrop.
	`soft light`	Darkens or lightens the colors, depending on the source color value. The effect is similar to shining a diffused spotlight on the backdrop.
	`difference`	Subtracts the darker of the two constituent colors from the lighter color. Painting with white inverts the backdrop color; painting with black produces no change.
	`exclusion`	Produces an effect similar to that of the Difference mode but lower in contrast. Painting with white inverts the backdrop color; painting with black produces no change.
	`hue`	Creates a color with the hue of the source color and the saturation and luminosity of the backdrop color.

	`saturation`	Creates a color with the saturation of the source color and the hue and luminosity of the backdrop color. Painting with this mode in an area of the backdrop that is a pure gray (no saturation) produces no change.
	`color`	Creates a color with the hue and saturation of the source color and the luminosity of the backdrop color. This preserves the gray levels of the backdrop and is useful for coloring monochrome images or tinting color images.
	`luminosity`	Creates a color with the luminosity of the source color and the hue and saturation of the backdrop color. This produces an inverse effect to that of the Color mode.

23.4 Fadings

For complicated graphics, uniform transparency settings are not always sufficient. Suppose, for instance, that while you paint a picture, you want the transparency to vary smoothly from completely opaque to completely transparent. This is a "shading-like" transparency. For such a form of transparency I will use the term *fading* (as a noun). They are also known as *soft masks*, *opacity masks*, *masks*, or *soft clips*.

23.4.1 Creating Fadings

How do we specify a fading? This is a bit of an art since the underlying mechanism is quite powerful, but a bit difficult to use.

Let us start with a bit of terminology. A *fading* specifies for each point of an area the transparency of that point. This transparency can by any number between 0 and 1. A *fading picture* is a normal graphic that, in a way to be described in a moment, determines the transparency of points inside the fading. Each fading has an underlying fading picture.

The fading picture is a normal graphic drawn using any of the normal graphic drawing commands. A fading and its fading picture are related as follows: Given any point of the fading, the transparency of this point is determined by the luminosity of the fading picture at the same position. The luminosity of a point determines "how bright" the point is. The brighter the point in the fading picture, the more opaque is the point in the fading. In particular, a white point of the fading picture is completely opaque in the fading and a black point of the fading picture is completely transparent in the fading. (The background of the fading picture is always transparent in the fading as if the background were black.)

It is rather counter-intuitive that a *white* pixel of the fading picture will be *opaque* in the fading and a *black* pixel will be *transparent*. For this reason, Ti*k*Z defines a color called `transparent` that is the same as `black`. The nice thing about this definition is that the color `transparent!`⟨*percentage*⟩ in the fading picture yields a pixel that is ⟨*percentage*⟩ percent transparent in the fading.

Turning a fading picture into a normal picture is achieved using the following commands, which are *only defined in the library*, namely the library `fadings`. So, to use them, you have to say `\usetikzlibrary{fadings}` first.

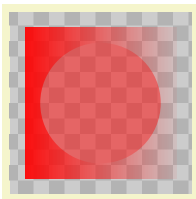
`\begin{tikzfadingfrompicture}[`⟨*options*⟩`]`
⟨*environment contents*⟩
`\end{tikzfadingfrompicture}`

This command works like a `{tikzpicture}`, only the picture is not shown, but instead a fading is defined based on this picture. To set the name of the picture, use the `name` option (which is normally used to set the name of a node).

`/tikz/name=`{⟨*name*⟩} (no default)

Use this option with the `{tikzfadingfrompicture}` environment to set the name of the fading. You *must* provide this option.

The following shading is 2cm by 2cm and gets more and more transparent from left to right, but is 50% transparent for a large circle in the middle.

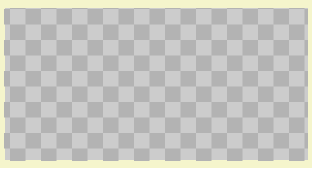


```
\usetikzlibrary {fadings,patterns}
\begin{tikzfadingfrompicture}[name=fade right with circle]
  \shade[left color=transparent!0,
         right color=transparent!100] (0,0) rectangle (2,2);
  \fill[transparent!50] (1,1) circle [radius=0.7];
\end{tikzfadingfrompicture}

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \pattern [pattern=checkerboard,pattern color=black!30]
                  (-1.2,-1.2) rectangle (1.2,1.2);

  \fill [path fading=fade right with circle,red] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

In the next example we create a fading picture that contains some text. When the fading is used, we only see the shading "through it".



```
\usetikzlibrary {fadings,patterns}
\begin{tikzfadingfrompicture}[name=tikz]
  \node [text=transparent!20,
         font=\fontencoding{T1}\fontfamily{ptm}\fontsize{45}{45}\bfseries]
    {Ti\emph{k}Z};
\end{tikzfadingfrompicture}

% Now we use the fading in another picture:
\begin{tikzpicture}
  \fill [black!20] (-2,-1) rectangle (2,1);
  \pattern [pattern=checkerboard,pattern color=black!30]
                  (-2,-1) rectangle (2,1);

  \shade[path fading=tikz,fit fading=false,
         left color=blue,right color=black]
    (-2,-1) rectangle (2,1);
\end{tikzpicture}
```

The same effect can also be achieved using knockout groups, see Section 23.5.

`\tikzfadingfrompicture`[⟨*options*⟩]
⟨*environment contents*⟩
`\endtikzfadingfrompicture`

The plain TEX version of the environment.

`\starttikzfadingfrompicture`[⟨*options*⟩]
⟨*environment contents*⟩
`\stoptikzfadingfrompicture`

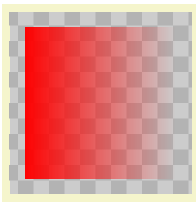
The ConTEXt version of the environment.

`\tikzfading`[⟨*options*⟩]

This command is used to define a fading similarly to the way a shading is defined. In the ⟨*options*⟩ you should

1. use the `name=`⟨*name*⟩ option to set a name for the fading,
2. use the `shading` option to set the name of the shading that you wish to use,
3. extra options for setting the colors of the shading (typically you will set them to the color `transparent!`⟨*percentage*⟩).

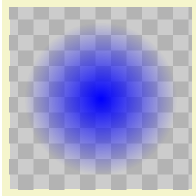
Then, a new fading named ⟨*name*⟩ will be created based on the shading.



```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade right,
            left color=transparent!0,
            right color=transparent!100]

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
                  (-1.2,-1.2) rectangle (1.2,1.2);

  \fill [red,path fading=fade right] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```



```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade out,
            inner color=transparent!0,
            outer color=transparent!100]

% Now we use the fading in another picture:
\begin{tikzpicture}
  % Background
  \fill [black!20] (-1.2,-1.2) rectangle (1.2,1.2);
  \path [pattern=checkerboard,pattern color=black!30]
                  (-1.2,-1.2) rectangle (1.2,1.2);

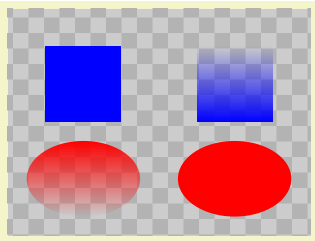
  \fill [blue,path fading=fade out] (-1,-1) rectangle (1,1);
\end{tikzpicture}
```

23.4.2 Fading a Path

A fading specifies for each pixel of a certain area how transparent this pixel will be. The following options are used to install such a fading for the current scope or path.

`/tikz/path fading=`⟨*name*⟩ (default scope's setting)

This option tells TikZ that the current path should be faded with the fading ⟨*name*⟩. If no ⟨*name*⟩ is given, the ⟨*name*⟩ set for the whole scope is used. Similarly to options like `draw` or `fill`, this option is reset for each path, so you have to add it to each path that should be faded. You can also specify `none` as ⟨*name*⟩, in which case fading for the path will be switched off in case it has been switched on by previous options or styles.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}[path fading=south]
  % Checker board
  \fill [black!20] (0,0) rectangle (4,3);
  \pattern [pattern=checkerboard,pattern color=black!30]
                  (0,0) rectangle (4,3);

  \fill [color=blue]                    (0.5,1.5) rectangle +(1,1);
  \fill [color=blue,path fading=north] (2.5,1.5) rectangle +(1,1);

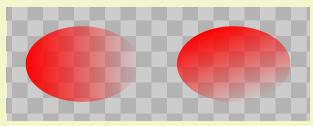
  \fill [color=red,path fading]  (1,0.75) ellipse [x radius=.75, y radius=.5];
  \fill [color=red]              (3,0.75) ellipse [x radius=.75, y radius=.5];
\end{tikzpicture}
```

`/tikz/fit fading=`⟨*boolean*⟩ (default `true`, initially `true`)

When set to `true`, the fading is shifted and resized (in exactly the same way as a shading) so that it covers the current path. When set to `false`, the fading is only shifted so that it is centered on the path's center, but it is not resized. This can be useful for special-purpose fadings, for instance when you use a fading to "punch out" something.

`/tikz/fading transform=`⟨*transformation options*⟩ (no default)

The ⟨*transformation options*⟩ are applied to the fading before it is used. For instance, if ⟨*transformation options*⟩ is set to `rotate=90`, the fading is rotated by 90 degrees.



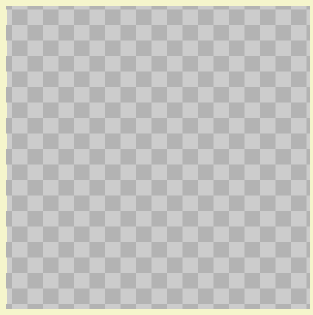
```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}[path fading=fade down]
  % Checker board
  \fill [black!20] (0,0) rectangle (4,1.5);
  \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,1.5);

  \fill [red,path fading,fading transform={rotate=90}]
    (1,0.75) ellipse [x radius=.75, y radius=.5];
  \fill [red,path fading,fading transform={rotate=30}]
    (3,0.75) ellipse [x radius=.75, y radius=.5];
\end{tikzpicture}
```

/tikz/fading angle=⟨*degree*⟩ (no default)

A shortcut for `fading transform={rotate=`⟨*degree*⟩`}`.

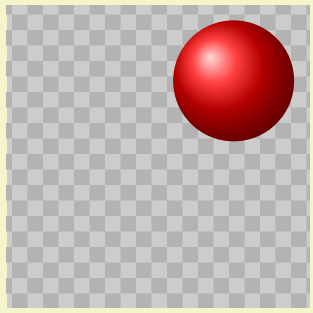
Note that you can "fade just about anything". In particular, you can fade a shading.



```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  % Checker board
  \fill [black!20] (0,0) rectangle (4,4);
  \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);

  \shade [ball color=blue,path fading=south] (2,2) circle [radius=1.8];
\end{tikzpicture}
```

The `fade inside` of the following example is more transparent in the middle than on the outside.

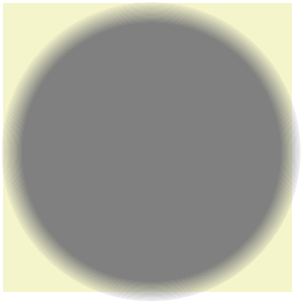


```
\usetikzlibrary {fadings,patterns}
\tikzfading[name=fade inside,
            inner color=transparent!80,
            outer color=transparent!30]
\begin{tikzpicture}
  % Checker board
  \fill [black!20] (0,0) rectangle (4,4);
  \path [pattern=checkerboard,pattern color=black!30] (0,0) rectangle (4,4);

  \shade [ball color=red] (3,3) circle [radius=0.8];
  \shade [ball color=white,path fading=fade inside] (2,2) circle [radius=1.8];
\end{tikzpicture}
```

Note that adding the `path fading` option to a node fades the (background) path, not the text itself. To fade the text, you need to use a scope fading (see below).

Note that using fadings in conjunction with patterns can create visually rather pleasing effects:



```
\usetikzlibrary {fadings,patterns,shadows}
\tikzfading[name=middle,
            top color=transparent!50,
            bottom color=transparent!50,
            middle color=transparent!20]
\begin{tikzpicture}
  \node       [circle,circular drop shadow,
               pattern=horizontal lines dark blue,
               path fading=south,
               minimum size=3.6cm] {};
  \pattern    [path fading=north,
               pattern=horizontal lines dark gray]
    (0,0) circle [radius=1.8cm];
  \pattern    [path fading=middle,
               pattern=crosshatch dots light steel blue]
    (0,0) circle [radius=1.8cm];
\end{tikzpicture}
```

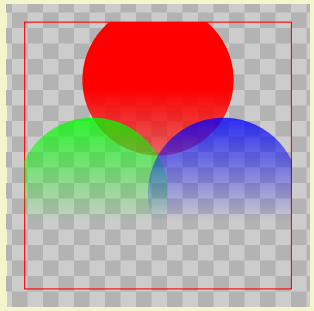
23.4.3 Fading a Scope

In addition to fading individual paths, you may also wish to "fade a scope", that is, you may wish to install a fading that is used globally to specify the transparency for all objects drawn inside a scope. This effect can also be thought of as a "soft clip" and it works in a similar way: You add the `scope fading` option to a path in a scope – typically the first one – and then all subsequent drawings in the scope are faded. You will use a `transparency group` in conjunction, see the end of this section.

`/tikz/scope fading`=⟨*fading*⟩ (no default)

In principle, this key works in exactly the same way as the `path fading` key. The only difference is, that the effect of the fading will persist after the current path till the end of the scope. Thus, the ⟨*fading*⟩ is applied to all subsequent drawings in the current scope, not just to the current path. In this regard, the option works very much like the `clip` option. (Note, however, that, unlike the `clip` option, fadings to not accumulate unless a transparency group is used.)

The keys `fit fading` and `fading transform` have the same effect as for `path fading`. Also that, just as for `path fading`, providing the `scope fading` option with a `{scope}` only sets the name of the fading to be used. You have to explicitly provide the `scope fading` with a path to actually install a fading.



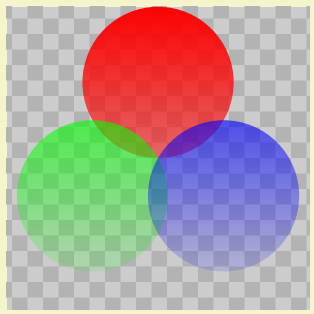
```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  \fill [black!20] (-2,-2) rectangle (2,2);
  \pattern [pattern=checkerboard,pattern color=black!30]
                   (-2,-2) rectangle (2,2);

  % The bounding box of the shading:
  \draw [red] (-50bp,-50bp) rectangle (50bp,50bp);

  \path [scope fading=south,fit fading=false] (0,0);
  % fading is centered at its natural size

  \fill[red]   ( 90:1) circle [radius=1];
  \fill[green] (210:1) circle [radius=1];
  \fill[blue]  (330:1) circle [radius=1];
\end{tikzpicture}
```

In the following example we resize the fading to the size of the whole picture:

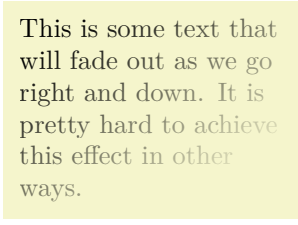


```
\usetikzlibrary {fadings,patterns}
\begin{tikzpicture}
  \fill [black!20] (-2,-2) rectangle (2,2);
  \pattern [pattern=checkerboard,pattern color=black!30]
                   (-2,-2) rectangle (2,2);

  \path [scope fading=south] (-2,-2) rectangle (2,2);

  \fill[red]   ( 90:1) circle [radius=1];
  \fill[green] (210:1) circle [radius=1];
  \fill[blue]  (330:1) circle [radius=1];
\end{tikzpicture}
```

Scope fadings are also needed if you wish to fade a node.




```
\usetikzlibrary {fadings}
\tikz \node [scope fading=south,fading angle=45,text width=3.5cm]
{
  This is some text that will fade out as we go right
  and down. It is pretty hard to achieve this effect in
  other ways.
};
```

23.5 Transparency Groups

Consider the following cross and sign. They "look wrong" because we can see how they were constructed, while this is not really part of the desired effect.



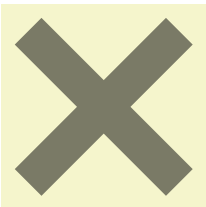
```
\begin{tikzpicture}[opacity=.5]
  \draw [line width=5mm] (0,0) -- (2,2);
  \draw [line width=5mm] (2,0) -- (0,2);
\end{tikzpicture}
```



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

  \node [opacity=.5]
        at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

Transparency groups are used to render them correctly:



```
\begin{tikzpicture}[opacity=.5]
  \begin{scope}[transparency group]
    \draw [line width=5mm] (0,0) -- (2,2);
    \draw [line width=5mm] (2,0) -- (0,2);
  \end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

  \begin{scope}[opacity=.5,transparency group]
    \node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
      {Smoking};
  \end{scope}
\end{tikzpicture}
```

`/tikz/transparency group=[`⟨*options*⟩`]` (no default)

This option can be given to a `scope`. It will have the following effect: The scope's contents is stroked / filled "ignoring any outside transparency". This means, all previous transparency settings are ignored (you can still set transparency inside the group, but never mind). For instance, in the forbidden sign example, the whole sign is first painted (conceptually) like the image on the left hand side. Note that some pixels of the sign are painted multiple times (up to three times), but only the last color "wins".

Then, when the scope is finished, it is painted as a whole. The *fill* transparency settings are now applied to the resulting picture. For instance, the pixel that has been painted three times is just red at the end, so this red color will be blended with whatever is "behind" the group on the page.



```
\usetikzlibrary {patterns,shapes.symbols}
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15](-1,-1) rectangle (3,1);
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

  \begin{scope}[transparency group,opacity=.5]
    \node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
      {Smoking};
  \end{scope}
\end{tikzpicture}
```

Note that in the example, the `opacity=.5` is not active inside the transparency group: The group is only established at beginning of the scope and all options given to the `{scope}` environment are set

before the group is established. To change the opacity *inside* the group, you need to open another scope inside it or use the `opacity` key with a command inside the group:



```
\usetikzlibrary {patterns,shapes.symbols}
\begin{tikzpicture}
  \pattern[pattern=checkerboard,pattern color=black!15](-1,-1) rectangle (3,1);
  \node at (0,0) [forbidden sign,line width=2ex,draw=red,fill=white] {Smoking};

  \begin{scope}[transparency group,opacity=.5]
    \node (s) at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
    {Smoking};

    \draw [opacity=.5, line width=2ex, blue] (1.2,0) -- (2.8,0);
  \end{scope}
\end{tikzpicture}
```

The ⟨*options*⟩ are a list of comma-separated options:

- `knockout` When this option is given inside the ⟨*options*⟩, the group becomes a so-called *knockout* group. This means, essentially, that inside the group everything is painted as if the "opacity" of a line or area were just another color channel. In particular, if you paint a pixel with opacity 0 inside a knockout group, this pixel becomes perfectly transparent immediately. In contrast, painting a pixel with something of opacity 0 normally has no effect.

  Not all renderers, let alone printers, will support this. At the time of writing, Apple's Preview will not show the following correctly (you should see the text Ti*k*Z in the middle):



```
\begin{tikzpicture}
  \shade [left color=red,right color=blue] (-2,-1) rectangle (2,1);
  \begin{scope}[transparency group=knockout]
    \fill [white] (-1.9,-.9) rectangle (1.9,.9);
    \node [opacity=0,
           font=\fontencoding{T1}\fontfamily{ptm}\fontsize{45}{45}\bfseries]
      {Ti\emph{k}Z};
  \end{scope}
\end{tikzpicture}
```

  In the example, we first draw a large shading and then, inside the transparency group "overwrite" most of this shading by a big white rectangle. The interesting part is the text of the node, which has opacity `0`. Normally, this would mean that nothing is shown. However, in a knockout group, we "paint" the text with an "opacity zero" color. The effect is that part of the totally opaque white rectangle gets overwritten by a perfectly transparent area (namely exactly the area taken up by the pixels of the text). When this whole knockout group is then placed on top of the shading, the shading will "shine through" at the knocked-out pixels.

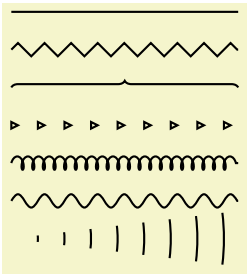
- `isolated=false` A group can be isolated or not. By default, they are isolated, since this is typically what you want. For details on what isolated groups are, exactly, see Section 7.3.4 of the PDF Specification, version 1.7.

Note that when a transparency group is created, Ti*k*Z must correctly determine the size of the material inside the group. Usually, this is no problem, but when you use things like `overlay` or `transform canvas`, trouble may result. In this case, please consult Section 115 on how to sidestep this problem in such cases.

24 Decorated Paths

24.1 Overview

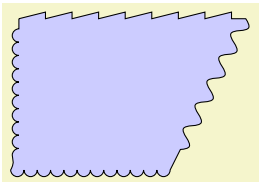
Decorations are a general concept to make (sub)paths “more interesting”. Before we have a look at the details, let us have a look at some examples:



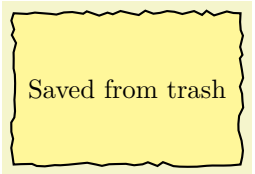
```
\usetikzlibrary { decorations.pathmorphing, decorations.pathreplacing, decorations.shapes,
}
\begin{tikzpicture}[thick]
  \draw                                                (0,3)   -- (3,3);
  \draw[decorate,decoration=zigzag]                    (0,2.5) -- (3,2.5);
  \draw[decorate,decoration=brace]                     (0,2)   -- (3,2);
  \draw[decorate,decoration=triangles]                 (0,1.5) -- (3,1.5);
  \draw[decorate,decoration={coil,segment length=4pt}] (0,1)   -- (3,1);
  \draw[decorate,decoration={coil,aspect=0}]           (0,.5)  -- (3,.5);
  \draw[decorate,decoration={expanding waves,angle=7}] (0,0)   -- (3,0);
\end{tikzpicture}
```




```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \node [fill=red!20,draw,decorate,decoration={bumps,mirror},
         minimum height=1cm]
  {Bumpy};
\end{tikzpicture}
```



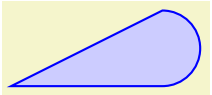
```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \filldraw[fill=blue!20]                    (0,3)
  decorate [decoration=saw]              { -- (3,3) }
  decorate [decoration={coil,aspect=0}] { -- (2,1) }
  decorate [decoration=bumps]            { -| (0,3) };
\end{tikzpicture}
```




```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \node [fill=yellow!50,draw,thick, minimum height=2cm, minimum width=3cm,
         decorate, decoration={random steps,segment length=3pt,amplitude=1pt}]
    {Saved from trash};
\end{tikzpicture}
```

The general idea of decorations is the following: First, you construct a path using the usual path construction commands. The resulting path is, in essence, a series of straight and curved lines. Instead of directly using this path for filling or drawing, you can then specify that it should form the basis for a decoration. In this case, depending on which decoration you use, a new path is constructed “along” the path you specified. For instance, with the `zigzag` decoration, the new path is a zigzagging line that goes along the old path.

Let us have a look at an example: In the first picture, we see a path that consists of a line, an arc, and a line. In the second picture, this path has been used as the basis of a decoration.

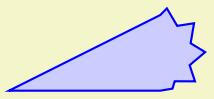


```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill
  [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```



```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill [decorate,decoration={zigzag}]
  [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
```

It is also possible to decorate only a subpath (the exact syntax will be explained later in this section).



```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill [decoration={zigzag}]
  [fill=blue!20,draw=blue,thick] (0,0) -- (2,1)
    decorate { arc (90:-90:.5) } -- cycle;
```

The `zigzag` decoration will be called a *path morphing* decoration because it morphs a path into a different, but topologically equivalent path. Not all decorations are path morphing; rather there are three kinds of decorations.

1. The just-mentioned *path morphing* decorations morph the path in the sense that what used to be a straight line might afterwards be a squiggly line or might have bumps. However, a line is still and a line and path deforming decorations do not change the number of subpaths.

   Examples of such decorations are the `snake` or the `zigzag` decoration. Many such decorations are defined in the library `decorations.pathmorphing`.

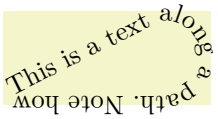
2. *Path replacing* decorations completely replace the path by a different path that is only "loosely based" on the original path. For instance, the `crosses` decoration replaces a path by a path consisting of a sequence of crosses. Note how in the following example filling the path has no effect since the path consist only of (numerous) unconnected straight line subpaths:

   ```
   \usetikzlibrary {decorations.shapes}
   \tikz \fill [decorate,decoration={crosses}]
     [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
   ```

   Examples of path replacing decorations are `crosses` or `ticks` or `shape backgrounds`. Such decorations are defined in the library `decorations.pathreplacing`, but also in `decorations.shapes`.

3. *Path removing* decorations completely remove the to-be-decorated path. Thus, they have no effect on the main path that is being constructed. Instead, they typically have numerous *side effects*. For instance, they might "write some text" along the (removed) path or they might place nodes along this path. Note that for such decorations the path usage command for the main path have no influence on how the decoration looks like.

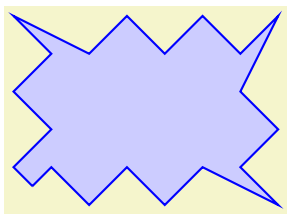

   ```
   \usetikzlibrary {decorations.text}
   \tikz \fill [decorate,decoration={text along path,
                  text=This is a text along a path. Note how the path is lost.}]
     [fill=blue!20,draw=blue,thick] (0,0) -- (2,1) arc (90:-90:.5) -- cycle;
   ```

Decorations are defined in different decoration libraries, see Section 50 for details. It is also possible to define your own decorations, see Section 103, but you need to use the PGF basic layer and a bit of theory is involved.

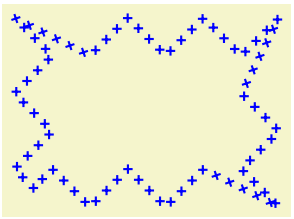
Decorations can be used to decorate already decorated paths. In the following three graphics, we start with a simple path, then decorate it once, and then decorate the decorated path once more.



```
\tikz \fill [fill=blue!20,draw=blue,thick]
  (0,0) rectangle (3,2);
```



```
\usetikzlibrary {decorations.pathmorphing}
\tikz \fill [fill=blue!20,draw=blue,thick]
  decorate[decoration={zigzag,segment length=10mm,amplitude=2.5mm}]
    { (0,0) rectangle (3,2) };
```



```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\tikz \fill [fill=blue!20,draw=blue,thick]
  decorate[decoration={crosses,segment length=2mm}] {
    decorate[decoration={zigzag,segment length=10mm,amplitude=2.5mm}] {
      (0,0) rectangle (3,2)
    }
  };
```

One final word of warning: Decorations can be pretty slow to typeset and they can be inaccurate. The reason is that PGF has to do a *lot* of rather difficult computations in the background and TEX is not very good at doing math. Decorations are fastest when applied to straight line segments, but even then they are much slower than other alternatives. For instance, the `ticks` decoration can be simulated by clever use of a dashing pattern and the dashing pattern will literally be thousands of times faster to typeset. However, for most decorations there are no real alternatives.

TikZ Library `decorations`

```
\usetikzlibrary{decorations} % LaTeX and plain TeX
\usetikzlibrary[decorations] % ConTeXt
```

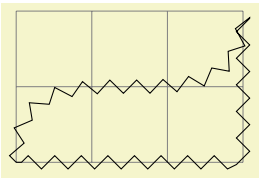
In order to use decorations, you first have to load a `decorations` library. This `decorations` library defines the basic options described in the following, but it does not define any new decorations. This is done by libraries like `decorations.text`. Since these more specialized libraries include the `decorations` library automatically, you usually do not have to bother about it.

24.2 Decorating a Subpath Using the Decorate Path Command

The most general way to decorate a (sub)path is the following path command.

`\path` … `decorate[`⟨*options*⟩`]{`⟨*subpath*⟩`}` … ;

This path operation causes the ⟨*subpath*⟩ to be decorated using the current decoration. Depending on the decoration, this may or may not extend the current path.

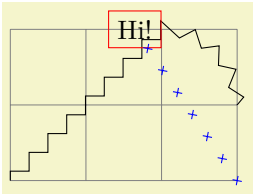


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration={name=zigzag}]
        { (0,0) .. controls (0,2) and (3,0) .. (3,2) |- (0,0) };
\end{tikzpicture}
```

The path can include straight lines, curves, rectangles, arcs, circles, ellipses, and even already decorated paths (that is, you can nest applications of the `decorate` path command, see below).

Due to the limits on the precision in TEX, some inaccuracies in positioning when crossing input segment boundaries may occasionally be found.

You can use nodes normally inside the ⟨*subpath*⟩.




```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration={name=zigzag}]
    { (0,0) -- (2,2) node (hi) [left,draw=red] {Hi!} arc(90:0:1)};

  \draw [blue] decorate [decoration={crosses}] {(3,0) -- (hi)};
\end{tikzpicture}
```

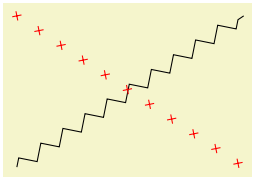
The following key is used to select the decoration and also to select further "rendering options" for the decoration.

`/pgf/decoration=`⟨*decoration options*⟩ (no default)

alias `/tikz/decoration`

This option is used to specify which decoration is used and how it will look like. Note that this key will *not* cause any decorations to be applied, immediately. It takes the `decorate` path command or

the `decorate` option to actually decorate a path. The `decoration` option is only used to specify which decoration should be used, in principle. You can also use this option at the beginning of a picture or a scope to specify the decoration to be used with each invocation of the `decorate` path command. Naturally, any local options of the `decorate` path command override these "global" options.



```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\begin{tikzpicture}[decoration=zigzag]
  \draw       decorate                        {(0,0) -- (3,2)};
  \draw [red] decorate [decoration=crosses] {(0,2) -- (3,0)};
\end{tikzpicture}
```

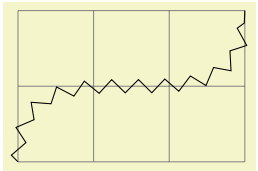
The ⟨*decoration options*⟩ are special options (which have the path prefix `/pgf/decoration/`) that determine the properties of the decoration. Which options are appropriate for a decoration strongly depend on the decoration, you will have to look up the appropriate options in the documentation of the decoration, see Section 50.

There is one option (available only in Ti*k*Z) that is special:

`/pgf/decoration/name=`⟨*name*⟩ (no default, initially `none`)

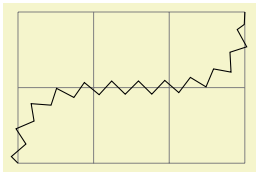
Use this key to set which decoration is to be used. The ⟨*name*⟩ can both be a decoration or a meta-decoration (you need to worry about the difference only if you wish to define your own decorations).

If you set ⟨*name*⟩ to `none`, no decorations are added.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration={name=zigzag}]
        { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
\end{tikzpicture}
```

Since this option is used so often, you can also leave out the `name=` part. Thus, the above example can be rewritten more succinctly:

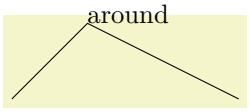


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw decorate [decoration=zigzag]
        { (0,0) .. controls (0,2) and (3,0) .. (3,2) };
\end{tikzpicture}
```

In general, when ⟨*decoration options*⟩ are parsed, for each unknown key it is checked whether that key happens to be a (meta-)decoration and, if so, the `name` option is executed for this key.

Further options allow you to adjust the position of decorations relative to the to-be-decorated path. See Section 24.4 below for details.

Recall that some decorations actually completely remove the to-be-decorated path. In such cases, the construction of the main path is resumed after the `decorate` path command ends.

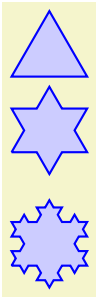



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text along path,text=
      around and around and around and around we go}]

  \draw (0,0) -- (1,1) decorate { -- (2,1) } -- (3,0);
\end{tikzpicture}
```

It is permissible to nest `decorate` commands. In this case, the path resulting from the first decoration process is used as the to-be-decorated path for the second decoration process. This is especially useful for drawing fractals. The `Koch snowflake` decoration replaces a straight line like _____ by _/_.

Repeatedly applying this transformation to a triangle yields a fractal that looks a bit like a snowflake, hence the name.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch snowflake,draw=blue,fill=blue!20,thick]
  \filldraw (0,0) -- ++(60:1) -- ++(-60:1) -- cycle ;
  \filldraw decorate{ (0,-1) -- ++(60:1) -- ++(-60:1) -- cycle };
  \filldraw decorate{ decorate{ (0,-2.5) -- ++(60:1) -- ++(-60:1) -- cycle }};
\end{tikzpicture}
```

24.3 Decorating a Complete Path

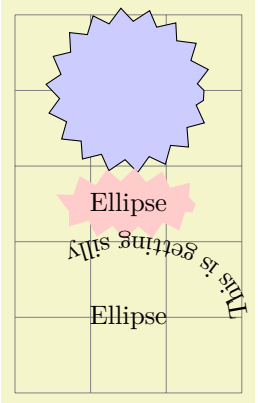
You may sometimes wish to decorate a path over whose construction you have no control. For instance, the path of the background of a node is created without having a chance to issue a `decorate` path command. In such cases you can use the following option, which allows you to decorate a path "after the fact".

`/tikz/decorate=`⟨*boolean*⟩ (default `true`)

When this key is set, the whole path is decorated after it has been finished. The decoration used for decorating the path is set via the `decoration` way, in exactly the same way as for the `decorate` path command. Indeed, the following two commands have the same effect:

1. `\path decorate[`⟨*options*⟩`] {`⟨*path*⟩`};`
2. `\path [decorate,`⟨*options*⟩`] `⟨*path*⟩`;`

The main use or the `decorate` option is the you can also use it with the nodes. It then causes the background path of the node to be decorated. Note that you can decorate a background path only once in this manner. That is, in contrast to the `decorate` path command you cannot apply this option twice (this would just set it to `true`, once more).



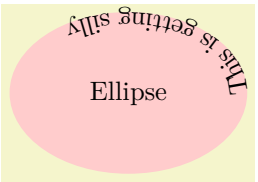

```
\usetikzlibrary { decorations.pathmorphing, decorations.text, shapes.geometric, }
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] (0,0) grid (3,5);

  \draw [fill=blue!20,decorate] (1.5,4) circle [radius=1cm];

  \node at (1.5,2.5) [fill=red!20,decorate,ellipse] {Ellipse};

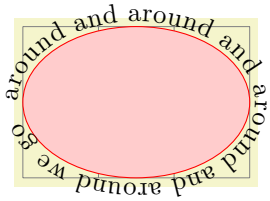
  \node at (1.5,1) [inner sep=6mm,fill=red!20,decorate,ellipse,decoration=
    {text along path,text={This is getting silly}}] {Ellipse};
\end{tikzpicture}
```

In the last example, the `text along path` decoration removes the path. In such cases it is useful to use a pre- or postaction to cause the decoration to be applied only before or after the main path has been used. Incidentally, this is another application of the `decorate` option that you cannot achieve with the decorate path command.




```
\usetikzlibrary { decorations.pathmorphing, decorations.text, shapes.geometric, }
\begin{tikzpicture}[decoration=zigzag]
  \node at (1.5,1) [inner sep=6mm,fill=red!20,ellipse,
    postaction={decorate,decoration=
    {text along path,text={This is getting silly}}}] {Ellipse};
\end{tikzpicture}
```

Here is more useful example, where a postaction is used to add the path after the main path has been drawn.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\fill [draw=red,fill=red!20,
        postaction={decorate,decoration={raise=2pt,text along path,
          text=around and around and around and around we go}}]
  (0,1) arc (180:-180:1.5cm and 1cm);
\end{tikzpicture}
```

24.4 Adjusting Decorations

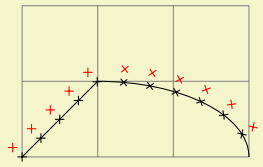
24.4.1 Positioning Decorations Relative to the To-Be-Decorate Path

The following option, which are only available with Ti*k*Z, allow you to modify the positioning of decorations relative to the to-be-decorated path.

`/pgf/decoration/raise`=⟨*dimension*⟩ (no default, initially `0pt`)

The segments of the decoration are raised by ⟨*dimension*⟩ relative to the to-be-decorated path. More precisely, the segments of the path are offset by this much "to the left" of the path as we travel along the path. This raising is done after and in addition to any transformations set using the `transform` option (see below).

A negative ⟨*dimension*⟩ will offset the decoration "to the right" of the to-be-decorated path.

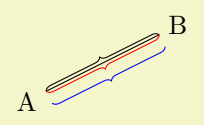


```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw (0,0) -- (1,1) arc (90:0:2 and 1);
  \draw      decorate [decoration=crosses]
        { (0,0) -- (1,1) arc (90:0:2 and 1) };
  \draw[red] decorate [decoration={crosses,raise=5pt}]
        { (0,0) -- (1,1) arc (90:0:2 and 1) };
\end{tikzpicture}
```

`/pgf/decoration/mirror`=⟨*boolean*⟩ (no default)

Causes the segments of the decoration to be mirrored along the to-be-decorated path. This is done after and in addition to any transformations set using the `transform` and/or `raise` options.

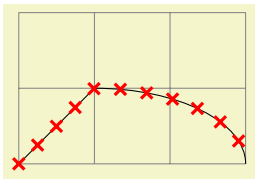



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}
  \node (a)          {A};
  \node (b) at (2,1) {B};
  \draw                                                  (a) -- (b);
  \draw[decorate,decoration=brace]                       (a) -- (b);
  \draw[decorate,decoration={brace,mirror},red]          (a) -- (b);
  \draw[decorate,decoration={brace,mirror,raise=5pt},blue] (a) -- (b);
\end{tikzpicture}
```

`/pgf/decoration/transform`=⟨*transformations*⟩ (no default)

This key allows you to specify general ⟨*transformations*⟩ to be applied to the segments of a decoration. These transformations are applied before and independently of `raise` and `mirror` transformations. The ⟨*transformations*⟩ should be normal Ti*k*Z transformations like `shift` or `rotate`.

In the following example the `shift only` transformation is used to make sure that the crosses are *not* sloped along the path.



```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \draw (0,0) -- (1,1) arc (90:0:2 and 1);
  \draw[red,very thick] decorate [decoration={
              crosses,transform={shift only},shape size=1.5mm}]
        { (0,0) -- (1,1) arc (90:0:2 and 1) };
\end{tikzpicture}
```

24.4.2 Starting and Ending Decorations Early or Late

You sometimes may wish to "end" a decoration a bit early on the path. For instance, you might wish a `snake` decoration to stop 5mm before the end of the path and to continue in a straight line. There are different ways of achieving this effect, but the easiest may be the `pre` and `post` options, which only have an effect in Ti*k*Z. Note, however, that they can only be used with decorations, not with meta-decorations.

`/pgf/decoration/pre`=⟨*decoration*⟩ (no default, initially `lineto`)

This key sets a decoration that should be used before the main decoration starts. The ⟨*decoration*⟩ will be used for a length of `pre length`, which `0pt` by default. Thus, for the `pre` option to have any effect, you also need to set the `pre length` option.



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=lineto,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

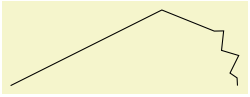


```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=moveto,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

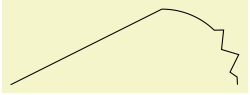


```
\usetikzlibrary { decorations.pathmorphing, decorations.shapes, }
\tikz [decoration={zigzag,pre=crosses,pre length=1cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

Note that the default `pre` option is `lineto`, not `curveto`. This means that the default `pre` decoration will not follow curves (for efficiency reasons). Change the `pre` key to `curveto` if you have a curved path.



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```



```
\usetikzlibrary {decorations.pathmorphing}
\tikz [decoration={zigzag,pre=curveto,pre length=3cm}]
  \draw [decorate] (0,0) -- (2,1) arc (90:0:1);
```

`/pgf/decoration/pre length`=⟨*dimension*⟩ (no default, initially `0pt`)

This key sets the distance along which the pre-decoration should be used. If you do not need/wish a pre-decoration, set this key to `0pt` (exactly this string, not just to something that evaluates to the same things such as `0cm`).

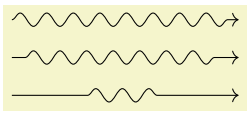
`/pgf/decorations/post`=⟨*decoration*⟩ (no default, initially `lineto`)

Works like `pre`, only for the end of the decoration.

/pgf/decorations/post length=⟨*dimension*⟩ (no default, initially `0pt`)

Works like `pre length`, only for the end of the decoration.

Here is a typical example that shows how these keys can be used:



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
  [decoration=snake,
   line around/.style={decoration={pre length=#1,post length=#1}}]

  \draw[->,decorate]                   (0,0)    -- ++(3,0);
  \draw[->,decorate,line around=5pt]   (0,-5mm) -- ++(3,0);
  \draw[->,decorate,line around=1cm]   (0,-1cm) -- ++(3,0);
\end{tikzpicture}
```

25 Transformations

PGF has a powerful transformation mechanism that is similar to the transformation capabilities of METAFONT. The present section explains how you can access it in Ti*k*Z.

25.1 The Different Coordinate Systems

It is a long process from a coordinate like, say, $(1, 2)$ or $(1\text{cm}, 5\text{pt})$, to the position a point is finally placed on the display or paper. In order to find out where the point should go, it is constantly "transformed", which means that it is mostly shifted around and possibly rotated, slanted, scaled, and otherwise mutilated.

In detail, (at least) the following transformations are applied to a coordinate like $(1, 2)$ before a point on the screen is chosen:

1. PGF interprets a coordinate like $(1, 2)$ in its xy-coordinate system as "add the current x-vector once and the current y-vector twice to obtain the new point".
2. PGF applies its coordinate transformation matrix to the resulting coordinate. This yields the final position of the point inside the picture.
3. The backend driver (like `dvips` or `pdftex`) adds transformation commands such that the coordinate is shifted to the correct position in TEX's page coordinate system.
4. PDF (or PostScript) apply the canvas transformation matrix to the point, which can once more change the position on the page.
5. The viewer application or the printer applies the device transformation matrix to transform the coordinate to its final pixel coordinate on the screen or paper.

In reality, the process is even more involved, but the above should give the idea: A point is constantly transformed by changes of the coordinate system.

In Ti*k*Z, you only have access to the first two coordinate systems: The xy-coordinate system and the coordinate transformation matrix (these will be explained later). PGF also allows you to change the canvas transformation matrix, but you have to use commands of the core layer directly to do so and you "better know what you are doing" when you do this. The moment you start modifying the canvas matrix, PGF immediately loses track of all coordinates and shapes, anchors, and bounding box computations will no longer work.

25.2 The XY- and XYZ-Coordinate Systems

The first and easiest coordinate systems are PGF's xy- and xyz-coordinate systems. The idea is very simple: Whenever you specify a coordinate like `(2,3)` this means $2v_x + 3v_y$, where v_x is the current x-*vector* and v_y is the current y-*vector*. Similarly, the coordinate `(1,2,3)` means $v_x + 2v_y + 3v_z$.

Unlike other packages, PGF does not insist that v_x actually has a y-component of 0, that is, that it is a horizontal vector. Instead, the x-vector can point anywhere you want. Naturally, *normally* you will want the x-vector to point horizontally.

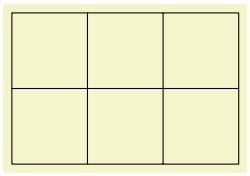
One undesirable effect of this flexibility is that it is not possible to provide mixed coordinates as in $(1, 2\text{pt})$. Life is hard.

To change the x-, y-, and z-vectors, you can use the following options:

`/tikz/x=`⟨*value*⟩ (no default, initially `1cm`)

If ⟨*value*⟩ is a dimension, the x-vector of PGF's xyz-coordinate system is set up to point ⟨*value*⟩ to the right, that is, to $(\langle value\rangle, 0pt)$.


```
\begin{tikzpicture}
  \draw                  (0,0)   -- +(1,0);
  \draw[x=2cm,color=red] (0,0.1) -- +(1,0);
\end{tikzpicture}
```



```
\tikz \draw[x=1.5cm] (0,0) grid (2,2);
```

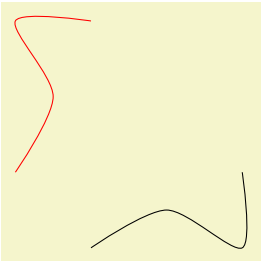
The last example shows that the size of steppings in grids, just like all other dimensions, are not affected by the x-vector. After all, the x-vector is only used to determine the coordinate of the upper right corner of the grid.

If ⟨*value*⟩ is a coordinate, the x-vector of PGF's xyz-coordinate system is set to the specified coordinate. If ⟨*value*⟩ contains a comma, it must be put in braces.



```
\begin{tikzpicture}
  \draw                                (0,0) -- (1,0);
  \draw[x={(2cm,0.5cm)},color=red] (0,0) -- (1,0);
\end{tikzpicture}
```

You can use this, for example, to exchange the meaning of the x- and y-coordinate.



```
\begin{tikzpicture}[smooth]
  \draw plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
  \draw[x={(0cm,1cm)},y={(1cm,0cm)},color=red]
        plot coordinates{(1,0) (2,0.5) (3,0) (3,1)};
\end{tikzpicture}
```

/tikz/y=⟨*value*⟩ (no default, initially `1cm`)

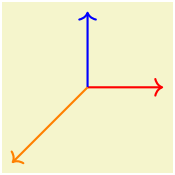
Works like the `x=` option, only if ⟨*value*⟩ is a dimension, the resulting vector points to $(0, \langle value \rangle)$.

/tikz/z=⟨*value*⟩ (no default, initially -3.85mm)

Works like the `y=` option, but now a dimension is the point $(\langle value \rangle, \langle value \rangle)$.

The default z-axis vector $(-3.85\text{mm}, -3.85\text{mm})$ behaves as a shear to make the z part of coordinates visible when projected onto the flat page.

A z-axis vector of $(0, 0)$, e.g., `z=0pt` gives a z-axis pointing directly out of the page. This is good if you're applying your own rotations to construct and then view a 3D object from some angle (view in the sense of project the rotated parts onto the page as the XY plane).



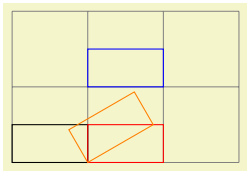
```
\begin{tikzpicture}[z=-1cm,->,thick]
  \draw[color=red] (0,0,0) -- (1,0,0);
  \draw[color=blue] (0,0,0) -- (0,1,0);
  \draw[color=orange] (0,0,0) -- (0,0,1);
\end{tikzpicture}
```

25.3 Coordinate Transformations

PGF and Ti*k*Z allow you to specify *coordinate transformations*. Whenever you specify a coordinate as in `(1,0)` or `(1cm,1pt)` or `(30:2cm)`, this coordinate is first "reduced" to a position of the form "x points to the right and y points upwards". For example, `(1in,5pt)` is reduced to "$72\frac{27}{100}$ points to the right and 5 points upwards" and `(90:100pt)` means "0pt to the right and 100 points upwards".

The next step is to apply the current *coordinate transformation matrix* to the coordinate. For example, the coordinate transformation matrix might currently be set such that it adds a certain constant to the x value. Also, it might be set up such that it, say, exchanges the x and y value. In general, any "standard"

transformation like translation, rotation, slanting, or scaling or any combination thereof is possible. (Internally, PGF keeps track of a coordinate transformation matrix very much like the concatenation matrix used by PDF or PostScript.)



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) rectangle (1,0.5);
  \begin{scope}[xshift=1cm]
    \draw             [red]    (0,0) rectangle (1,0.5);
    \draw[yshift=1cm] [blue]   (0,0) rectangle (1,0.5);
    \draw[rotate=30]  [orange] (0,0) rectangle (1,0.5);
  \end{scope}
\end{tikzpicture}
```

The most important aspect of the coordinate transformation matrix is *that it applies to coordinates only!* In particular, the coordinate transformation has no effect on things like the line width or the dash pattern or the shading angle. In certain cases, it is not immediately clear whether the coordinate transformation matrix *should* apply to a certain dimension. For example, should the coordinate transformation matrix apply to grids? (It does.) And what about the size of arced corners? (It does not.) The general rule is: "If there is no 'coordinate' involved, even 'indirectly', the matrix is not applied.". However, sometimes, you simply have to try or look it up in the documentation whether the matrix will be applied.

Setting the matrix cannot be done directly. Rather, all you can do is to "add" another transformation to the current matrix. However, all transformations are local to the current TEX-group. All transformations are added using graphic options, which are described below.

Transformations apply immediately when they are encountered "in the middle of a path" and they apply only to the coordinates on the path following the transformation option.

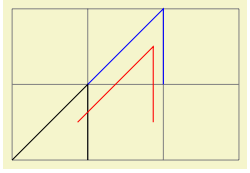


```
\tikz \draw (0,0) rectangle (1,0.5) [xshift=2cm] (0,0) rectangle (1,0.5);
```

A final word of warning: You should refrain from using "aggressive" transformations like a scaling of a factor of 10 000. The reason is that all transformations are done using TEX, which has a fairly low accuracy. Furthermore, in certain situations it is necessary that TikZ *inverts* the current transformation matrix and this will fail if the transformation matrix is badly conditioned or even singular (if you do not know what singular matrices are, you are blessed).

`/tikz/shift={`⟨*coordinate*⟩`}` (no default)

Adds the ⟨*coordinate*⟩ to all coordinates.

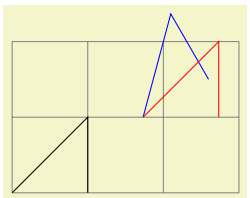


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                        (0,0) -- (1,1) -- (1,0);
  \draw[shift={(1,1)},blue]    (0,0) -- (1,1) -- (1,0);
  \draw[shift={(30:1cm)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

`/tikz/shift only` (no value)

This option does not take any parameter. Its effect is to cancel all current transformations except for the shifting. This means that the origin will remain where it is, but any rotation around the origin or scaling relative to the origin or skewing will no longer have an effect.

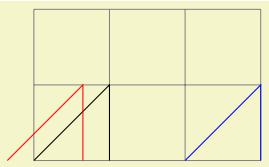
This option is useful in situations where a complicated transformation is used to "get to a position", but you then wish to draw something "normal" at this position.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                                        (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,blue]             (0,0) -- (1,1) -- (1,0);
  \draw[rotate=30,xshift=2cm,shift only,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/xshift=⟨*dimension*⟩ (no default)

Adds ⟨*dimension*⟩ to the x value of all coordinates.



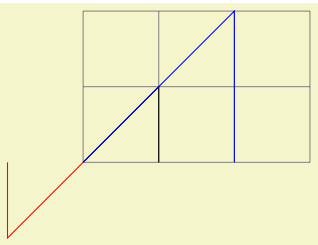
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                  (0,0) -- (1,1) -- (1,0);
  \draw[xshift=2cm,blue]  (0,0) -- (1,1) -- (1,0);
  \draw[xshift=-10pt,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yshift=⟨*dimension*⟩ (no default)

Adds ⟨*dimension*⟩ to the y value of all coordinates.

/tikz/scale=⟨*factor*⟩ (no default)

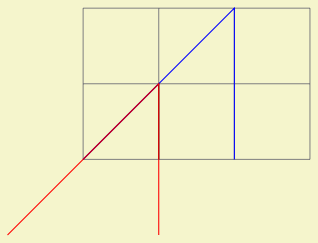
Multiplies all coordinates by the given ⟨*factor*⟩. The ⟨*factor*⟩ should not be excessively large in absolute terms or very close to zero.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw               (0,0) -- (1,1) -- (1,0);
  \draw[scale=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[scale=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/scale around={⟨*factor*⟩:⟨*coordinate*⟩} (no default)

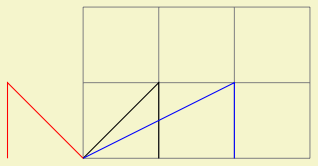
Scales the coordinate system by ⟨*factor*⟩, with the "origin of scaling" centered on ⟨*coordinate*⟩ rather than the origin.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                              (0,0) -- (1,1) -- (1,0);
  \draw[scale=2,blue]                (0,0) -- (1,1) -- (1,0);
  \draw[scale around={2:(1,1)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/xscale=⟨*factor*⟩ (no default)

Multiplies only the x-value of all coordinates by the given ⟨*factor*⟩.



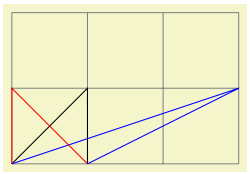
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw               (0,0) -- (1,1) -- (1,0);
  \draw[xscale=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[xscale=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yscale=⟨*factor*⟩ (no default)

Multiplies only the y-value of all coordinates by ⟨*factor*⟩.

/tikz/xslant=⟨*factor*⟩ (no default)

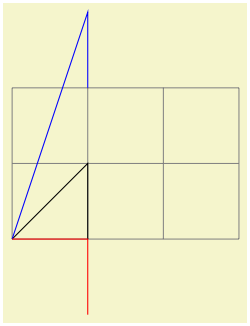
Slants the coordinate horizontally by the given ⟨*factor*⟩:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                (0,0) -- (1,1) -- (1,0);
  \draw[xslant=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[xslant=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/yslant=⟨*factor*⟩ (no default)

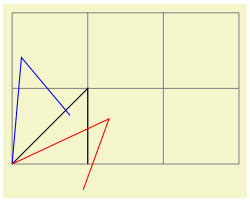
Slants the coordinate vertically by the given ⟨*factor*⟩:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                (0,0) -- (1,1) -- (1,0);
  \draw[yslant=2,blue] (0,0) -- (1,1) -- (1,0);
  \draw[yslant=-1,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/rotate=⟨*degree*⟩ (no default)

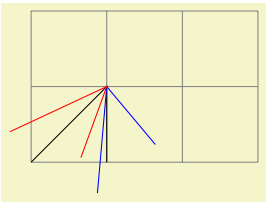
Rotates the coordinate system by ⟨*degree*⟩:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                 (0,0) -- (1,1) -- (1,0);
  \draw[rotate=40,blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate=-20,red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/rotate around={⟨*degree*⟩:⟨*coordinate*⟩} (no default)

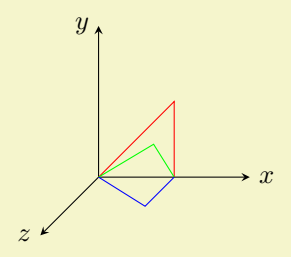
Rotates the coordinate system by ⟨*degree*⟩ around the point ⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                                  (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={40:(1,1)},blue] (0,0) -- (1,1) -- (1,0);
  \draw[rotate around={-20:(1,1)},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/rotate around x=⟨*angle*⟩ (no default)

This key sets the x, y and z vectors of the PGF xyz-coordinate system so that they are rotated by ⟨*angle*⟩ around the axis corresponding to the x-vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.

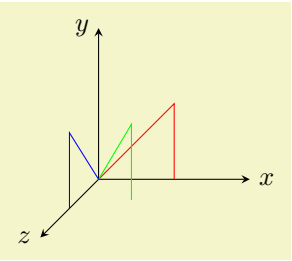



```
\begin{tikzpicture}[>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left]  {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left]  {$z$};

  \draw [red,   rotate around x=0]  (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [green, rotate around x=45] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [blue,  rotate around x=90] (0,0,0) -- (1,1,0) -- (1,0,0);
\end{tikzpicture}
```

/tikz/rotate around y=⟨*angle*⟩ (no default)

This key sets the x, y and z vectors of the PGF xyz-coordinate system so that they are rotated by ⟨*angle*⟩ around the axis corresponding to the y-vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.

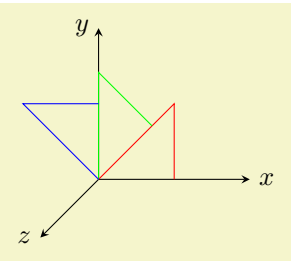



```
\begin{tikzpicture}[>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left]  {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left]  {$z$};

  \draw [red,   rotate around y=0]   (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [green, rotate around y=-45] (0,0,0) -- (1,1,0) -- (1,0,0);
  \draw [blue,  rotate around y=-90] (0,0,0) -- (1,1,0) -- (1,0,0);
\end{tikzpicture}
```

/tikz/rotate around z=⟨*angle*⟩ (no default)

This key sets the x, y and z vectors of the PGF xyz-coordinate system so that they are rotated by ⟨*angle*⟩ around the axis corresponding to the z-vector. The rotation is applied so that when looking towards the origin along this axis, positive angles result in an anticlockwise rotation.

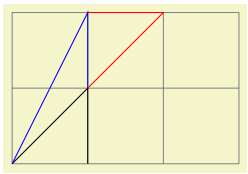



```
\begin{tikzpicture}[>=stealth]
  \draw [->] (0,0,0) -- (2,0,0) node [at end, right] {$x$};
  \draw [->] (0,0,0) -- (0,2,0) node [at end, left]  {$y$};
  \draw [->] (0,0,0) -- (0,0,2) node [at end, left]  {$z$};

  \draw [red,   rotate around z=0]   (0,0) -- (1,1) -- (1,0);
  \draw [green, rotate around z=45]  (0,0) -- (1,1) -- (1,0);
  \draw [blue,  rotate around z=90]  (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/cm={⟨*a*⟩,⟨*b*⟩,⟨*c*⟩,⟨*d*⟩,⟨*coordinate*⟩} (no default)

applies the following transformation to all coordinates: Let (x, y) be the coordinate to be transformed and let ⟨*coordinate*⟩ specify the point (t_x, t_y). Then the new coordinate is given by $\left(\begin{smallmatrix} a & c \\ b & d \end{smallmatrix}\right)\left(\begin{smallmatrix} x \\ y \end{smallmatrix}\right) + \left(\begin{smallmatrix} t_x \\ t_y \end{smallmatrix}\right)$. Usually, you do not use this option directly.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                                (0,0) -- (1,1) -- (1,0);
  \draw[cm={1,1,0,1,(0,0)},blue]       (0,0) -- (1,1) -- (1,0);
  \draw[cm={0,1,1,0,(1cm,1cm)},red]    (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

/tikz/reset cm (no value)

Completely resets the coordinate transformation matrix to the identity matrix. This will destroy not only the transformations applied in the current scope, but also all transformations inherited from surrounding scopes. Do not use this option, unless you really, really know what you are doing.

25.4 Canvas Transformations

A *canvas transformation*, see Section 99.4 for details, is best thought of as a transformation in which the drawing canvas is stretched or rotated. Imaging writing something on a balloon (the canvas) and then blowing air into the balloon: Not only does the text become larger, the thin lines also become larger. In particular, if you scale the canvas by a factor of two, all lines are twice as thick.

Canvas transformations should be used with great care. In most circumstances you do *not* want line widths to change in a picture as this creates visual inconsistency.

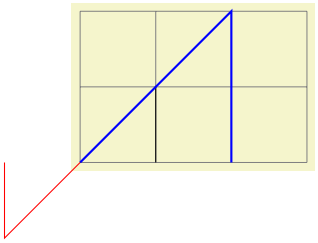
Just as important, when you use canvas transformations *PGF loses track of positions of nodes and of picture sizes* since it does not take the effect of canvas transformations into account when it computes coordinates of nodes (do not, however, rely on this; it may change in the future).

Finally, note that a canvas transformation always applies to a path as a whole, it is not possible (as for coordinate transformations) to use different transformations in different parts of a path.

In short, you should not use canvas transformations unless you really know what you are doing.

`/tikz/transform canvas=`⟨*options*⟩ (no default)

The ⟨*options*⟩ should contain coordinate transformations options like `scale` or `xshift`. Multiple options can be given, their effects accumulate in the usual manner. The effect of these ⟨*options*⟩ (immediately) changes the current canvas transformation matrix. The coordinate transformation matrix is not changed. Tracking of the picture size is (locally) switched off and the node coordinate will no longer be correct.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw                                    (0,0) -- (1,1) -- (1,0);
  \draw[transform canvas={scale=2},blue]   (0,0) -- (1,1) -- (1,0);
  \draw[transform canvas={rotate=180},red] (0,0) -- (1,1) -- (1,0);
\end{tikzpicture}
```

26 Animations

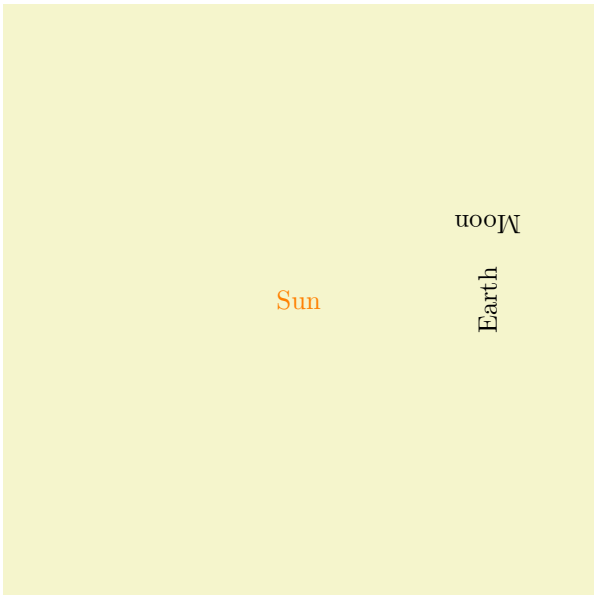
Ti*k*Z Library animations

```
\usetikzlibrary{animations} % LaTeX and plain TeX
\usetikzlibrary[animations] % ConTeXt
```

This library must be loaded in order to use animations with Ti*k*Z.

26.1 Introduction

An *animation* changes the appearance of some part of a graphic over time. The archetypical animation is, of course, a *movement* of some part of a picture, but a change of, say, the opacity of a path is also an animation. Ti*k*Z allows you to specify such animations using special keys and notations.

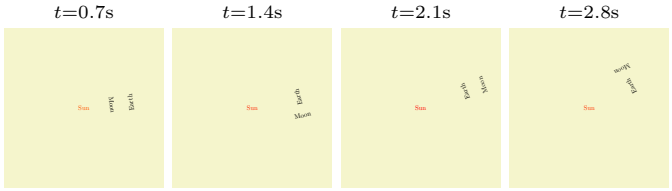



```
\usetikzlibrary {animations}
\begin{tikzpicture}[
    animate/orbit/.style 2 args = {
        myself:shift = {
          along = {
            (0,0) circle [radius=#1]
          } sloped in #2s/10,
          repeats }} ]

 \node :color = {0s = "orange",
                 2s = "red",
                 4s = "orange",
                 repeats}
       {Sun};

  \begin{scope}[animate={orbit={2.5cm}{365}}]
    \node {Earth};
    \node [animate={orbit={1cm}{28}}] {Moon};
  \end{scope}

  \useasboundingbox (-3.8,-3.8) (3.8,3.8);
\end{tikzpicture}
```



Adding an animation to a Ti*k*Z picture is done as follows:

1. *Before* or *in the options of* the to-be-animated object you specify the object together with an *attribute* that you wish to animate. Attributes are things like the fill color or the line width or the position of the object.
2. You specify *when* this attribute should have *which* values using a so-called *timeline*. This is just a curve that specifies for each point in time which value the attribute should have.
3. You can additionally use further options to configure the animation, for instance you can specify that the animation should repeat or that it should only start when a certain object is clicked.

As a simple example, let us move a circle within thirty seconds by three centimeters to the left:



```
\usetikzlibrary {animations}
\tikz \draw :xshift = {0s = "0cm", 30s = "-3cm", repeats} (0,0) circle [radius=5mm];
```

As can be seen, a special syntax is used in several places: Entries with a colon such as `:xshift` specify an attribute, values are specified in quotation marks. This syntax will be explained in more detail later on.

26.1.1 Animations Change Attributes

Before we plunge into the details of how animations are specified, it is important to understand what Ti*k*Z actually does when creating an animation: It does *not* (as all other animation packages do) precompute a sequence of pictures that are later somehow displayed in rapid succession. Neither does it insert an external video into the document. Rather, a Ti*k*Z animation is just an "annotation" in the output that a certain attribute of a certain object should change over time in some specific way when the object is displayed. It is the job of the document viewer application to actually compute and display the animation. The big advantage of this approach is that animations neither increase the output file sizes noticeably nor do they really slow down TEX: The hard and complicated calculations are done by the viewer application. The disadvantage is, of course, that a document viewer application must understand the annotations and actually compute and display the animations. The SVG format is a format for which this is possible, the popular PDF format is not. For the SVG format, there are actually different possible ways of "formulating" the animations (using SMIL or CSS or JavaScript) and they have different advantages and disadvantages.

To make a long story short: Ti*k*Z animations currently work only with SVG output (and use the SMIL "flavor" of describing animations). In future, it may well happen that other "flavor" of describing animations will be added, but it is very unlikely that PDF will ever support animations in a useful way.

It is, however, possible to create "snapshots" of an animation and insert these into PDF files (or any other kind of file including SVG files), see Section 26.6 for details. Snapshots are also useful for creating "printed versions" of animations and all of the small sequences of pictures in the manual that are used for showing what an animation key does have been creating using snapshots.

26.1.2 Limitations of the Animation System

There are a certain limitations of the animation system that you should keep in mind when considering how and when to use it:

1. As pointed out earlier, animations require a specific output format (currently only SVG is supported).
2. It is extremely difficult to animate "lines between moving nodes" correctly. Consider code like `\draw(a)--(b);` where `a` and `b` are nodes. Now, when you animate the position of `(a)`, the line connecting `(a)` and `(b)` will, unfortunately, not "move along" automatically (but it is easy to move the whole group of `(a)`, `(b)`, and the connecting line as whole). You must "cheat" and introduce some "virtual" nodes, which leads to rather complex and bloated code.
3. Animations are taken into consideration for bounding box computations, but only for shifts, not for rotations, scaling, or skewing and also possibly not when multiple shifts are active at the same time for the same object.

26.1.3 Concepts: (Graphic) Objects

During an animation an attribute of a certain "object" changes over time. The term "object" is deliberately a bit vague since there are numerous different "things" whose attributes can change. In detail, the following objects have attributes that can be animated:

1. Nodes, which are created by the `\node` command (and, also, internally by commands such as `\graph`). For nodes, different parts of the node can be animated separately; for instance, you can animate the color of the background path, but also the color of the text, and also the color of the foreground path (though most nodes do not have a foreground path) and also the color of different text parts (though only few nodes have multiple text parts).
2. Graphic scopes, which are created by numerous command, including the `{scope}` environment, the `\scopes` command, but also `\tikz` itself creates a graphic scope and so does each node and even each path.
3. View boxes, which can only be created using the `views` library.
4. Paths, which you create using the `\path` command or commands like `\draw` that call `\path` internally. However, the (usually background) path of a node can also be animated. Note that "animating the path" really means that the path itself should change over time; in essence, you can "warp" a path over time.

In all of these cases, you must either specify the animation inside the object's options using `animate` or use the `name` key to name the object and, then, refer to it in an `animate`. For nodes you can, of course, use the `(`⟨*node name*⟩`)` syntax to name the node. Recall that you must *always* specify the animation *before* the object is created; it is not possible to animate an already created object.

There is a special syntax for choosing the object of an animation, see Section 26.3.1, but you can also use the `object` key to choose them directly, see Section 26.2.3.

26.1.4 Concepts: Attributes

In addition to the to-be-animated object, you must also choose an *attribute* that you wish to animate. Attributes are things like the color of an object, the position, but also things like the line width. The syntax for choosing attributes and the list of attributes will be explained in detail later on.

Most attributes correspond directly to attributes that are directly supported by the backend driver (SVG), but this is not always the case. For instance, for a node, Ti*k*Z differentiates between the fill color, the draw (stroke) color, and the text color, while SVG treats the text color are a special case of the fill color. Ti*k*Z will do some internal mappings to ensure that you can animate the "Ti*k*Z attributes" even when they are not directly supported.

The same syntax that is used for specifying object is also used to specify attributes, see Section 26.3.1, but you could also set them directly using the `attribute` key see Section 26.2.4.

26.1.5 Concepts: Timelines

Once an object and an attribute have been chosen, a *timeline* needs to be established. This is, essentially, a curve that specifies for each "moment in time" which value the attribute should have.

A timeline has a *start* and an *end*, but the start need not be the "moment zero" (we will come to that) and may even be negative, while the end may be at infinity. You specify the timeline by specifying for certain points in time what the value is at that moment; for all other moments the value is then interpolated. For instance, if you specify that the attribute `:xshift` (the "horizontal position" of the object) is 0 mm at time 5 s and 10 mm at time 10 s, then at 7.5 s it will be 5 mm and at 9 s it will be 8 mm (assuming a linear interpolation). The resulting optical effect will be that the object *smoothly moves* by one centimeter to the right over a period of five seconds, starting five seconds after "moment zero".

Now, what is the "moment zero", the "beginning of an animation"? If nothing else is specified, an animation starts immediately when the graphic is shown and this is the moment zero relative to which the timeline is measured. However, it is also possible to change this. In particular, you can specify that the moment zero is when a particular *event* occurs such as the user clicking on another object or another animation ending or starting.

The interpolation of values is not always a straightforward affair. Firstly, for certain kinds of values is not clear how an interpolation should be computed. How does one interpolate between two paths? Between the colors red and green? Between the values `"true"` and `"false"`? In these cases, one must define carefully what the interpolation should be. Secondly, you may wish to use a non-linear interpolation, which is useful for "easing" motions: The visual effect of the movement specified above is that the object sits still from moment 0 for five seconds, then there is an "infinite acceleration" causing the object to suddenly move at the speed of 2 mm per second, then there is no acceleration at all for five seconds, causing the object to move for one centimeter, followed by an "infinite negative acceleration" that makes the object come to a full stop. As a viewer you experience these infinite accelerations as "unrealistic", spoiling the effect of watching a (virtual) physical process. Non-linear interpolations allow you to avoid this effect.

Just as for specifying objects and attributes, there is also a special syntax for specifying times and values.

26.2 Creating an Animation

26.2.1 The Animate Key

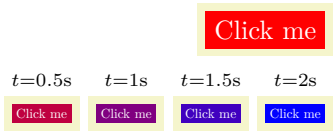
In order to animate a picture, you create timelines for all objects and attributes that change during the animation. The key `animate` is used for creating these timelines.

`/tikz/animate=`⟨*animation specification*⟩ (no default)

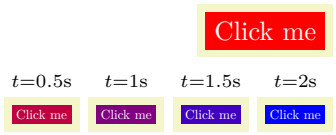
You must place all specifications of animations inside uses of `animate`. You can, and usually should, place the specification of all timelines of a single picture inside a single use of this key since it will reset the time and the fork time (explained in Section 26.2.6). You can, however, use this key several times, in principle. Note that if you animate the same attribute of the same object in two different uses of

`animate`, two separate timelines will result (and complicated rules are used to determine which one "wins" in case they specify conflicting values for the attribute at different times).

The key can be used at all places where a Ti*k*Z key is used; typically you will use it with a `{scope}` environment, inside the options of a node, or directly with the `\tikz` command:

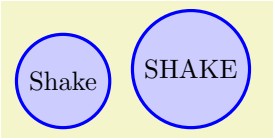



```
\usetikzlibrary {animations}
\tikz \node [fill, text = white, animate = {
  myself:fill = {0s = "red", 2s = "blue", begin on = click }}] {Click me};
```




```
\usetikzlibrary {animations}
\tikz [animate = {a node:fill = {0s = "red", 2s = "blue",
                                 begin on = click}}]
  \node (a node) [fill, text = white] {Click me};
```

The details of what, exactly, happens in the ⟨*animation specification*⟩ will be described in the rest of this section. However, basically, an ⟨*animation specification*⟩ is just a sequence of normal Ti*k*Z key–value pairs that get executed with the path prefix `/tikz/animate` and with some special syntax handlers installed. In particular, you can define styles for this key path and use them. For instance, we can define a `shake` animation like this:




```
\usetikzlibrary {animations}
\tikzset{
  animate/shake/.style = {myself:xshift = { begin on=click,
      0s = "0mm", 50ms = "#1", 150ms = "-#1", 250ms = "#1", 300ms = "0mm" }}}
\tikz \node [fill = blue!20, draw=blue, very thick, circle,
  animate = {shake = 1mm}] {Shake};
\tikz \node [fill = blue!20, draw=blue, very thick, circle,
  animate = {shake = 2mm}] {SHAKE};
```

Note that, as stressed earlier, you can only use the `animate` key to specify animations for objects that do not yet exist. The node and object names mentioned in a specification always refer to "upcoming" objects; already existing objects of the same name are not influenced.

You can use the `name` key inside `animate` to "name" the animation. Once named, you can later reference the animation in other animations; for instance, you can say that another animation should start when the present animation has ended.

26.2.2 Timeline Entries

The "job" of the options passed to the `animate` key is to specify the timelines of the animation of (a part of) a picture. For each object and each attribute there may or may not be a timeline and, if present, the timeline consist of sequences of pairs of times and values. Thus, the most basic entity of an animation specification is a tuple consisting of five parts, which are selected by five different keys:

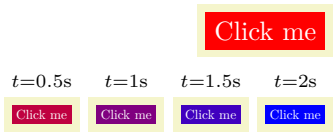
- `object` for selecting the object,
- `attribute` for selecting the attribute,
- `id` for selecting the timeline id (explained in Section 26.2.5),
- `time` for selecting a time, and
- `value` for selecting a value.

When all of these parts have been set up (using the above keys, which will be explained in more detail in a moment), you can use the following key to create an entry:

`/tikz/animate/entry` (no value)

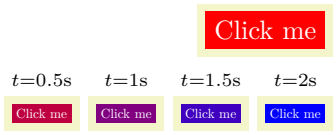
Each time this key is used in the options of `animate`, Ti*k*Z checks whether the five keys `object`, `attribute`, `id`, `time`, and `value` are set. If one of them is not set, nothing happens. (The `id` key is set to the value `default` by default, all other keys must be set explicitly.)

If all of these keys are set, a *time–value* pair is created and added to the timeline of attribute of the object. Additionally, all options starting with `/tikz/animate/options/`, which also influence the timeline like `begin on`, are also added to the timeline of the object–attribute pair.




```
\usetikzlibrary {animations}
\tikz [animate = {
  object = node, attribute = fill, time = 0s, value = red, entry,
  object = node, attribute = fill, time = 2s, value = blue, entry,
  object = node, attribute = fill, begin on = click, entry}]
  \node (node) [fill, text=white] { Click me };
```

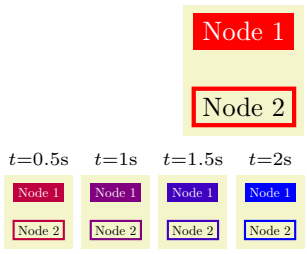
In the above example, it would not have been necessary the specify the object and the attribute in each line, they retain their values unless they are overwritten. Thus, we could also have written:




```
\usetikzlibrary {animations}
\tikz [animate = {
  object = node, attribute = fill, time = 0s, value = red, entry,
                                   time = 2s, value = blue, entry,
                                   begin on = click, entry}]
  \node (node) [fill, text=white] { Click me };
```

Note, however, that in both examples we actually add the time–value pair (2s, blue) twice since the `time` and `value` keys also retain their settings and, thus, for the third `entry` they have the same values as before and a new pair is added. While this superfluous pair is not a problem in the example (it has no visual effect), we will see later on how such pairs can be avoided by using the `scope` key.

A sequence of calls of `entry` can freely switch between objects and attributes (that is, between timelines), but the times for any given timeline must be given in non-decreasing order:




```
\usetikzlibrary {animations}
\tikz [animate = {
  object = node,  attribute = fill, time = 0s, value = red, entry,
  object = node2, attribute = draw, entry,
  object = node,  attribute = fill, time = 2s, value = blue, entry,
  object = node2, attribute = draw, entry,
  object = node,  attribute = fill, begin on = click, entry,
  object = node2, attribute = draw, begin on = click, entry}] {
  \node (node)  [fill, text=white]           { Node 1 };
  \node (node2) [draw, ultra thick] at (0,-1) { Node 2 };
}
```

In the above example, we could not have exchanged the first two lines of the `animate` options with the third and fourth line since the values for time `0s` must come before the values for time `2s`.

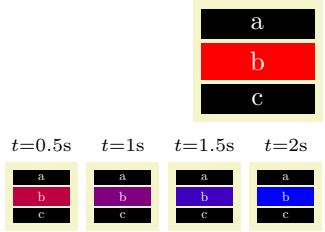
In the following, we have a closer look at the five keys the influence the `entry` key and then have a look at ways of grouping keys more easily.

26.2.3 Specifying Objects

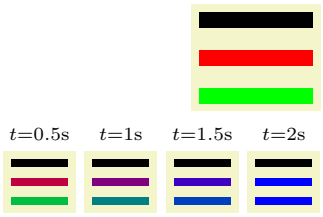
You use the `object` key to select the object(s) to which the next use of `entry` applies. There is also a special syntax for this, which is explained in Section 26.3.1.

`/tikz/animate/object`=⟨*list of objects*⟩ (no default)

The ⟨*list of objects*⟩ is a comma-separated list of strings of the form ⟨*object*⟩.⟨*type*⟩. All of the objects in the list are selected as to-be-animate object for the next use of the `entry` key. The objects referred to by ⟨*object*⟩ will be the *next* objects with the `name` key set to ⟨*object*⟩. You can apply the `name` key to nodes (where you can also use the special parentheses-syntax and put the name in parentheses, it has the same effect), but also to scopes and paths. (The `name path` key is not the same as `name`; it is an older key from the intersections package and not related.)




```
\usetikzlibrary {animations}
\tikz [animate = { object = b, :fill = {0s = "red", 2s = "blue",
                                        begin on = click }}] {
  \node (a) [fill, text = white, minimum width=1.5cm] at (0,1cm) {a};
  \node (b) [fill, text = white, minimum width=1.5cm] at (0,5mm) {b};
  \node (c) [fill, text = white, minimum width=1.5cm] at (0,0mm) {c}; }
```




```
\usetikzlibrary {animations}
\tikz [animate = { object = b, :fill = {0s = "red", 2s = "blue",
                                        begin on = click },
                   object = c, :fill = {0s = "green", 2s = "blue",
                                        begin on = click } }] {
  \scoped [name = a, yshift=1cm] \fill (0,0) rectangle (1.5cm,2mm);
  \scoped [name = b, yshift=5mm] \fill (0,0) rectangle (1.5cm,2mm);
  \scoped [name = c, yshift=0mm] \fill (0,0) rectangle (1.5cm,2mm); }
```

If the ⟨*object*⟩ name is never used later in the file, no animation is created.

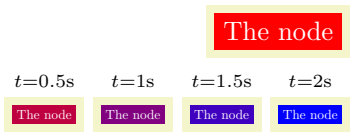
The ⟨*object*⟩ may also be the special text `myself`. In this case, the referenced object is the scope or object to which the `animate` key is given. If an object is named `myself` (as in `\node (myself) ...`), you cannot reference this node using the `object` key, `myself` *always* refers to the object where the `animate` key is given (of course, you can animate the node named `myself` by placing the `animate` key inside the options of this node; you only cannot "remotely" add an animation to it).

The ⟨*object*⟩ may be followed by a dot and a *type*. This is needed in rare cases where you want to animate only a special "part" of an object that is not accessible in other ways. Normally, Ti*k*Z takes care of choosing these types automatically, you only need to set these "if you know what you are doing".

26.2.4 Specifying Attributes

`/tikz/animate/attribute`=⟨*list of attributes*⟩ (no default)

The list of attributes must be a comma-separated list of attribute names. The timelines specified later will apply to all of these attributes (and to all objects previously selected using `object`). Possible attributes include colors, positions, line width, but even the paths themselves. The exact list of possible attributes is documented in Section 26.4.



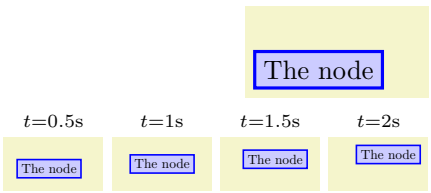

```
\usetikzlibrary {animations}
\tikz [animate = {attribute = fill, n: = { 0s = "red", 2s = "blue",
                                           begin on = click } }]
  \node (n) [fill, text = white] {The node};
```

26.2.5 Specifying IDs

`/tikz/animate/id`=⟨*id*⟩ (no default, initially `default`)

Timelines are use to defined how the values of an attribute of an object change over time. In many cases, you will have at most one timeline for each object–attribute pair, but, sometimes, you may wish to have more than one timeline for the same object and the same attribute. For instance, you might have a timeline that specifies a changing `shift` of a node in some direction and, at the same time, another timeline that specifies an additional `shift` in some other direction(s). The problem is that there is only one `shift` attribute and it would be difficult to compute the joint effect of the two timelines.

For this purpose, timelines are actually identified not only by the object–attribute pair but, in reality, by the triple consisting of the object, the attribute, and the value of this key. We can now specify two separate timelines:




```
\usetikzlibrary {animations}
\tikz [animate = {
  id = 1, n:shift = { 0s = "{(0,0)}", 2s = "{(0,5mm)}", begin on = click },
  id = 2, n:shift = { 0s = "{(0,0)}", 2s = "{(5mm,0)}", begin on = click }
}]
  \node (n) [fill = blue!20, draw=blue, very thick] {The node};
```

The default value of `id` is `default`.

Because of the possibility of creating multiple timelines for the same attribute, it may happen that there is more than one timeline active that is "trying to modify" a given attribute. In this case, the following rules are used to determine, which timeline "wins":

1. If no animation is active at the current time (all animation either have not yet started or they have already ended), then the `base` value given in the animation encountered last in the code is used. (If there are no base values, the attribute is taken from the surrounding scope and the animations have "no effect".)

2. If there are several active animations, the one that has started last is used and its value is used.

3. If there are several active animations that have started at the same time, the one that comes last in the code is used.

Note that these rules do not apply to transformations of the canvas since these are always additive (or, phrased differently, they are always all active and the effects accumulate).

26.2.6 Specifying Times

`/tikz/animate/time=`⟨*time*⟩`later` (no default)

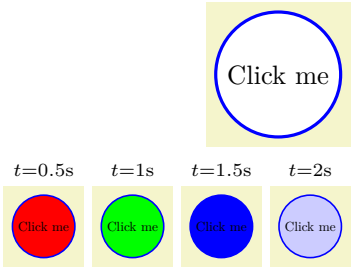
Sets the time for the next time–value pair in a call of `entry` to ⟨*time*⟩ plus the current fork time. The text `later` is optional. Both "fork times" and the optional `later` will be explained in a moment.

Time Parsing. The ⟨*time*⟩ is parsed using the command `\pgfparsetime`, which is essentially the same as the usual math parser of Ti*k*Z, and the result is interpreted as a time in seconds. Thus, a ⟨*time*⟩ of `2+3` means "5 seconds" and a ⟨*time*⟩ of `2*(2.1)` means "4.2 seconds". (You could even specify silly times like `1in`, which results in the time "72.27 seconds". Please do not do that.) The "essentially" refers to the fact that some extras are installed when the time parser is running:

- The postfix operator `s` is added, which has no effect. Thus, when you write `5s` you get the same results as `5`, which is exactly 5 seconds as desired.
- The postfix operator `ms` is added, which divides a number by 1000, so `2ms` equals 0.002s.
- The postfix operator `min` is added, which multiplies a number by 60.
- The postfix operator `h` is added, which multiplies a number by 3600.
- The infix operator `:` is redefined, so that it multiplies its first argument by 60 and adds the second. This implies that `1:20` equals 80s and `01:00:00` equals 3600s.
- The parsing of octal numbers is switched off to allow things like `01:08` for 68s.

Note that you cannot use the colon syntax for times in things like `01:20 = "0"` would (falsely) be interpreted as: "For the object named `01` and its attribute named `20`, do something." You can, however, use `01:20` in arguments to the `time` key, meaning that you would have to write instead: `time = 1:20, "0"`, possibly surround by a `scope`.

Relative Times. You can suffix a `time` key with "`later`". In this case, the ⟨*time*⟩ is interpreted as an offset to the time in the previous use of the time key:




```
\usetikzlibrary {animations}
\tikz \node :fill = { begin on = click,
    0s = "white",
    500ms later = "red",
    500ms later = "green",  % same as 1s   = "-5mm"
    500ms later = "blue"} % same as 1.5s = "-2.5mm"
  [fill=blue!20, draw=blue, very thick, circle] {Click me};
```

In reality, the offset is not taken to just any previous use of the `time` key, but to the most recent use of this key or of the `resume` key in the current local TEX scope. Here is an example:

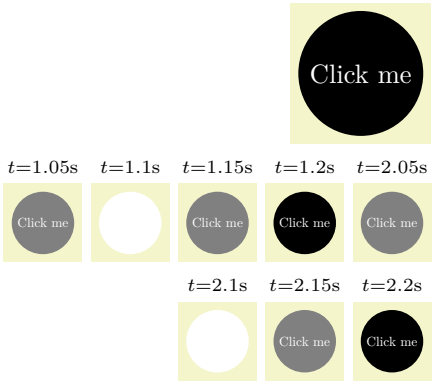
```
time = 2s,
time = 1s later,    % same as time = 3s
time = 500ms later, % same as time = 3.5s
time = 4s,
time = 1s later,    % same as time = 5s
scope = {           % opens a local scope
  time = 1s later,  % same as time = 6s
  time = 10s
  time = 1s later   % same as time = 11s
},                  % closes the scope, most recent time is 5s once more
time = 2s later     % same as time = 7s
```

Fork Times. The time meant by the value ⟨*time*⟩ passed to the `time` key is not used directly. Rather, Ti*k*Z adds the current *fork time* to it, which is `0s` by default. You can change the fork time using the following key:

`/tikz/animate/fork=`⟨*t*⟩ (default `0s later`)

Sets the fork time for the local scope to ⟨*t*⟩ and sets the current time to `0s`. In this scope, when you use "absolute" times like `0s` or `2s`, you actually refer to later times that have started as ⟨*t*⟩.

One application of forks is in the definition of keys that add a certain part to a longer animation. Consider for instance the definition of a `highlight` key:




```
\usetikzlibrary {animations}
\tikz [animate/highlight/.style = {
    scope = { fork = #1,
              :fill = { 0s = "black", 0.1s = "white", 0.2s = "black"} }
  }]
  \node [animate = { myself: = {
            :fill = { 0s = "black", begin on = click },
                      highlight = 1s, highlight = 2s } },
         fill = blue, text=white, very thick, circle] { Click me };
```

In the above example, we could also have written `0.1s later` instead of `0.2s` and, indeed, the whole style could have been defined using only times with `later`, eliminating the need for the `fork` key. However, using forks you can specify absolute times for things happening in a conceptual "subprocess" and also relative times. The name `fork` for the key is also borrowed from operating system theory, where a "fork" is the spawning of an independent process.

Remembering and Resuming Times. When you have a complicated animation with a long timeline, you will sometimes wish to start some animation when some other animation has reached a certain moment; but this moment is only reached through heavy use of `later` times and/or forks. In such situations, the following keys are useful:

`/tikz/animate/remember=`⟨*macroname*⟩ (no default)

This key stores the current time (the time of the last use of the `time` key) globally in the macro ⟨*macroname*⟩. This time will include the offset of the fork time:

```
time = 2s,
fork = 2s later,     % fork time is now 4s
time = 1s,           % local time is 1s, absolute time is 5s (1s + fork time)
time = 1s later,     % local time is 2s, absolute time is 6s (2s + fork time)
remember = \mytime   % \mytime is now 6s
```

`/tikz/animate/resume=`⟨*absolute time*⟩ (no default)

The ⟨*absolute time*⟩ is evaluated using `\pgfparsetime` and, then, the current time is set to the resulting time minus the fork time. When the ⟨*absolute time*⟩ is a macro previously set using `remember`, the net effect of this is that we return to the exact "moment" in the global time line when `remember` was used.

```
fork = 4s,
time = 1s,
remember = \mytime   % \mytime is now 5s
fork = 2s,           % fork time is now 2s, local time is 0s
resume   = \mytime   % fork time is still 2s, local time is 3s
```

Using resume you can easily implement a "join" operation for forked times. You simply remember the times at the ends of the forks and then resume the maximum time of these remembered times:

```
scope = {
  fork,
  time = 1s later,
  ...
  remember = \forka
},
scope = {
  fork,
  time = 5s later,
  ...
  remember = \forkb
},
scope = {
  fork,
  time = 2s later,
  ...
  remember = \forkc
},
resume = {max(\forka,\forkb,\forkc)} % "join" the three forks
```

26.2.7 Values

/tikz/animate/value=⟨*value*⟩ (no default)

This key sets the value of the next time–value pair created by `entry` to ⟨*value*⟩. The syntax of the ⟨*value*⟩ is not fixed, it depends on the type of the attribute. For instance, for an attribute like `opacity` the ⟨*value*⟩ must be an expression that can be evaluated to a number between 0 and 1; for the attribute `color` the ⟨*value*⟩ must, instead, be a color; and so on. Take care that when a value contains a comma, you must surround it by braces as in `"{(1,1)}"`.

The allowed texts for the ⟨*value*⟩ is always the same as the one you would pass to the Ti*k*Z option of the same name. For instance, since the Ti*k*Z option `shift` expects a coordinate, you use coordinates as ⟨*value*⟩ with the usual Ti*k*Z syntax (including all sorts of extensions, the animation system calls the standard Ti*k*Z parsing routines). The same is true of dimensions, scalar values, colors, and so on.

In addition to the values normally use for setting the attribute, you can also (sometimes) use the special text `current value` as ⟨*value*⟩. This means that the value of the point in the timeline should be whatever the value the attribute has at the beginning of the timeline. For instance, when you write

```
animate = { obj:color = { 0s = "current value", 2s = "white" } }
```

the color of `obj` will change from whatever color it currently has to white in two seconds. This is especially useful when several animations are triggered by user events and the current color of `obj` cannot be determined beforehand.

There are several limitations on the use of the text `current value`, which had to be imposed partly because of the limited support of this feature in SVG:

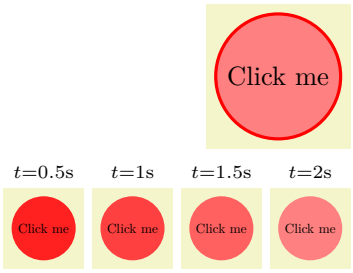
- You can use `current value` only with the first time in a timeline.
- You can only have two times in a timeline that starts with `current value`.
- You cannot use `current value` for timelines of which you wish to take a snapshot.

26.2.8 Scopes

When you specify multiple timelines at the same time, it is often useful and sometimes even necessary to have keys be set only locally. The following key makes this easy:

/tikz/animate/scope=⟨*options*⟩ (no default)

Executed the ⟨*options*⟩ inside a TEX scope. In particular, all settings made inside the scope have no effect after the end of the `scope`.




```
\usetikzlibrary {animations}
\tikz \node [animate = { myself: = { begin on = click,
    scope = { attribute = fill, repeats = 3, 0s = "red", 2s = "red!50" },
    scope = { attribute = draw,              0s = "red", 2s = "red!50" }
  }},
  fill=blue!20, draw=blue, very thick, circle] {Click me};
```

Without the use of the `scope` key, the `repeats` key would also affect the draw attribute.

While the `scope` key is useful for structuring timeline code, it also keeps the current time local to the scope, that is, if you use something like `1s later` after the scope, this will refer to one second after the last use of `time` *before* the scope. The times set inside the `scope` do not matter. While this is desirable effect for forks, you may also sometimes wish to synchronize the local time after the scope with the last time reached in the scope. The following key makes this easy:

`/tikz/animate/sync`=⟨*options*⟩ (no default)

A shorthand for `scope={` ⟨*options*⟩ `, remember=\temp},resume=\temp` where `\temp` is actually an internal name. The effect is that after a `sync` the local time just continues as if the scope where not present – but regarding everything else the effects are local to the `sync` scope.

26.3 Syntactic Simplifications

In the previous subsection we saw how timelines can be created by specifying the individual entries of the timelines sequentially. However, most of the time you will wish to use a simpler syntax that makes it easier to specify animations. This syntax is only available inside the `animate` key (it is switched on at the beginning) and consists of three "parts": The colon syntax, the time syntax, and the quote syntax.

26.3.1 The Colon Syntax I: Specifying Objects and Attributes

Inside the ⟨*animation specification*⟩ passed to the `animate` key, you can specify an object and an attribute of this object using the following syntax, whose use is detected by the presence of a colon inside a key:

⟨*object name(s)*⟩`:`⟨*attribute(s)*⟩ `={`⟨*options*⟩`}`

or

⟨*object name(s)*⟩`:`⟨*attribute(s)*⟩`_`⟨*id*⟩ `={`⟨*options*⟩`}`

In the place to the left of an equal sign, where you would normally use a key, you can instead place an object name and an attribute separated by a colon. Additionally, the attribute may be followed by an underscore and an ⟨*id*⟩, which identifies the timeline (see Section 26.2.5).

Each of these values may be missing, in which case it is not changed from its previous value.

The effect of the above code is the same as:

`sync = { object =` ⟨*objects*⟩`, attribute =` ⟨*attribute*⟩`, id =` ⟨*id*⟩`,` ⟨*options*⟩`, entry }`

although when the object, the attribute, or the id is left empty in the colon syntax, the corresponding setting will be missing in the above call of `sync`. Note that because of the `sync` the last time used inside the ⟨*options*⟩ will be available afterwards as the last time. Also note that an `entry` is added at the end, so any settings of keys like `begin` or `repeats` inside the ⟨*options*⟩ will get added to the timeline.

Let us now have a look at some examples. First, we set the ⟨*object name*⟩ to `mynode` and `othernode` and the ⟨*attribute*⟩ to `opacity` and to `color`:

```
animate = {
  mynode:opacity    = { 0s = "1",   5s = "0" },
  mynode:color      = { 0s = "red", 5s = "blue" },
  othernode:opacity = { 0s = "1",   5s = "0" },
}
```

Next, we do the same, but "in two steps": First, we set the object to `mynode`, but leave the attribute open and, then, set the attribute, but leave the object:

```
animate = {
  mynode: = {
    :opacity        = { 0s = "1",   5s = "0" },
    :color          = { 0s = "red", 5s = "blue" }
  },
  othernode:opacity = { 0s = "1",   5s = "0" },
}
```

Note how both in `mynode:` and in `:opacity` and `:color` you must provide the colon. Its presence signals that an object–attribute pair is being specified; only now either the object or the attribute is missing.

We can also do it the other way round:

```
animate = {
  :opacity = {
    mynode:         = { 0s = "1",   5s = "0" },
    othernode:      = { 0s = "1",   5s = "0" }
  },
  mynode:color      = { 0s = "red", 5s = "blue" }
}
```

Finally, if several objects should get the exact same values, we can also group them:

```
animate = {
  {mynode,othernode}:opacity = { 0s = "1",   5s = "0" },
  mynode:color               = { 0s = "red", 5s = "blue" }
}
```

As mentioned earlier, all references to objects will be interpreted to future objects, never to objects already created. Furthermore, also as mentioned earlier, Ti*k*Z allows you to specify `myself` as ⟨*object*⟩, which is interpreted as the scope or node where the `animate` is given (you cannot animate a node or scope named `myself`, this special name always refers to the current node). In order to have all attributes refer to the current object, you write:

```
\begin{scope} [animate = {
                 myself: = { % Animate the attribute of the scope
                   :opacity = { ... },
                   :xshift  = { ... }
                 }
               }]
  ...
\end{scope}
```

The list of permissible attributes is given in Section 26.4.

26.3.2 The Colon Syntax II: Animating Myself

A frequent use of the `animate` key is for animating attributes of the current object `myself`. In these cases, it is a bit length to write

```
[animate = { myself: = { :some attribute = {...} } } ]
```

in the options of a node or a scope. For this reason, Ti*k*Z allows you to use a special syntax with nodes and scopes:

1. In a ⟨*node specification*⟩, which is everything following a `node` command up to the content of the node (which is surrounded by curly braces), you can write

   `:some attribute = {`⟨*options*⟩`}`

   and this will have the same effect as if you had written

   `[animate = { myself: = { :some attribute = {`⟨*options*⟩`}}}]`

   Note that you can use this syntax repeatedly, but each use creates a new use of the `animate` key, resulting in a new timeline. In order to create complex timelines for several objects, use the `animate` key.

2. For the commands `\tikz`, `\scoped` and the environments `{tikzpicture}` and `{scope}`, when they are followed immediately by

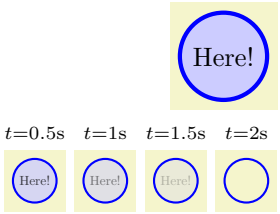
   `:some attribute = {`⟨*options*⟩`}`

then

`animate = { myself: = { :some attribute = {`⟨*options*⟩`}}}`

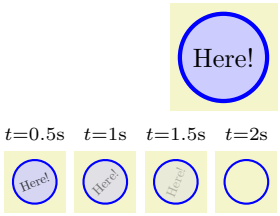
is added to the options of the command or scope. Again, you can use the syntax repeatedly. Note that when an opening square bracket is encountered, this special parsing stops.

Let us have a look at some examples. First, we use the syntax to set the fill opacity of a node:



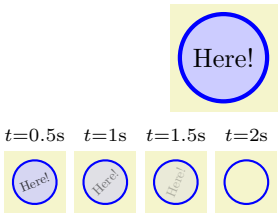

```
\usetikzlibrary {animations}
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Here!};
```

Next, we additionally rotate the node:



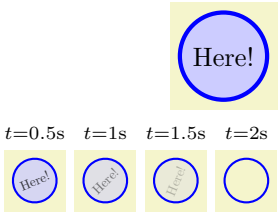

```
\usetikzlibrary {animations}
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  :rotate = { 0s="0", 2s="90", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Here!};
```

Note that there is no comma between consecutive uses of the colon syntax in this case. We could have exchanged the order of the options and the uses of the colon syntax:




```
\usetikzlibrary {animations}
\tikz \node
  :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle]
  :rotate = { 0s="0", 2s="90", begin on=click } {Here!};
```

We can also use the special syntax with the `\tikz` command itself:




```
\usetikzlibrary {animations}
\tikz  :fill opacity = { 0s="1", 2s="0", begin on=click }
       :rotate = { 0s="0", 2s="90", begin on=click }
       [ultra thick]
  \node [fill = blue!20, draw = blue, circle] {Here!};
```

Note that we could *not* have moved the `[ultra thick]` options before `:rotate` since the options in square brackets end the special parsing.

26.3.3 The Time Syntax: Specifying Times

For each object–attribute pair you must specify the *timeline* of the attribute. This is a curve that specifies for each "moment in time" which value the attribute should have. In the simplest case, you specify such a time–value pair as follows:

⟨*time*⟩ `="`⟨*value*⟩`"`

When you specify time–value pairs, you must specify the times in chronological order (so earlier times come first), but you may specify the same time several times (this is useful in situations where you have a "jump" from one value to another at a certain moment in time: you first specify the value "from which the attribute jumps" and then you specify the value "to which the attribute jumps" for the same moment).

The above syntax is just a special case of a more general situation. Let us start with the times. The general syntax for specifying times is as follows:

⟨*time*⟩ `=` ⟨*options*⟩

Here, ⟨*time*⟩ is a text that "looks like a time", which means that:

1. It is not a key and does not contain a colon and does not start with a quotation mark.
2. It starts with a digit, a plus or minus sing, a dot, or a parenthesis.

If these two things are the case, the above code is transformed to the following call:

`sync = {time =` ⟨*time*⟩`,` ⟨*options*⟩`, entry}`

26.3.4 The Quote Syntax: Specifying Values

We saw already in several examples that values are put in quotation marks (similar to the way this is done in XML). This quote syntax is as follows:

`"`⟨*value*⟩`"` `base =` ⟨*options*⟩

This syntax is triggered whenever a key starts with a quotation mark[5] (and note that when the ⟨*value*⟩ contains a comma, you have to surround it by curly braces *inside* the quotation marks as in `"{(1,1)}"`). Then, the following code is executed:

`sync = {value =` ⟨*value*⟩`,` ⟨*options*⟩`, entry}`

This means that when you write `1s = "red"`, what actually happens is that Ti*k*Z executes the following:

```
sync = { time = 1s, sync = { value = red, entry }, entry }
```

Note that the second entry has no effect since no value is specified and the `entry` key only "takes action" when both a time and a value have been specified. Thus, only the innermost `entry` does, indeed, create a time–value pair as desired.

In addition to the above, if you have added `base` after the closing quote, the following gets executed before the above `sync`:

`base = {value =` ⟨*value*⟩`}`

This makes it easy to specify base values for timelines.

Interestingly, instead of `1s="red"` you can also write `"red"=1s`. Let us now have a look at situations where this can be useful.

26.3.5 Timesheets

Using the `sync` key or using the three different syntactic constructs introduced earlier (the color syntax, the time syntax, the value syntax), you can organize the specification of an animation in different ways. Basically, the two most useful ways are the following:

1. You first select an object and an attribute for which you wish to establish a timeline and then provide the time–value pairs in a sequence:

```
animate = {
  obj:color = {
    0s = "red",
    2s = "blue",
    1s later = "green",
    1s later = "green!50!black",
    10s = "black"
  }
}
```

When you specify timelines for several attributes of the same object, you can group these together:

```
animate = {
  obj: = {
    :color = { 0s = "red", 2s = "green" },
    :opacity = { 0s = "1", 2s = "0" }
  }
}
```

In this way of specifying animations the "object comes first".

[5] Of catcode 12 for those knowledgeable of such things.

2. Alternatively, you can also group the animation by time and, for each "moment" (known as *keyframes*) you specify which values the attributes of the object(s) have:

```
animate = {
  0s = {
    obj:color = "red",
    obj:opacity = "1"
  },
  2s = {
    obj:color = "green",
    obj:opacity = "0"
  }
}
```

Naturally, in this case it would have been better to "move the object outside":

```
animate = {
  obj: = {
    0s = {
      :color = "red",
      :opacity = "1"
    },
    2s = {
      :color = "green",
      :opacity = "0"
    }
  }
}
```

When there are several objects involved, we can mix all of these approaches:

```
animate = {
  0s = {
    obj: = {
      :color = "red",
      :opacity = "1"
    },
    main node: = {
      :color = "black"
    }
  },
  2s = {
    obj: = {
      :color = "green",
      :opacity = "0"
    },
    main node: = {
      :color = "white"
    }
  }
}
```

26.4 The Attributes That Can Be Animated

The following ⟨*attributes*⟩ are permissible (actually, the attribute names do not include a colon, but since they will almost always be used with the colon syntax, it makes it easier to identify them):

- `:dash phase`
- `:dash pattern`
- `:dash`
- `:draw opacity`
- `:draw`
- `:fill opacity`
- `:fill`
- `:line width`

- `:opacity`
- `:position`
- `:path`
- `:rotate`
- `:scale`
- `:stage`
- `:text opacity`
- `:text`
- `:translate`
- `:view`
- `:visible`
- `:xscale`
- `:xshift`
- `:xskew`
- `:xslant`
- `:yscale`
- `:yshift`
- `:yskew`
- `:yslant`

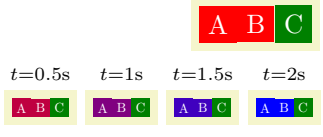
These attributes are detailed in the following sections, but here is a quick overview of those that do not have a Ti*k*Z key of the same name (and which thus do not just animate the attribute set using this key):

- `:shift` allows you to add an animated shifting of the canvas, just like Ti*k*Z's `shift` key. However, in conjunction with the `along` key, you can also specify the shifting along a path rather than via a timeline of coordinates.
- `:position` works similar to `:shift`, only the coordinates are not relative movements (no "shifts"), but refer to "absolute positions" in the picture.
- `:path` allows you to animate a path (it will morph). The "values" are now paths themselves.
- `:view` allows you to animate the view box of a view.
- `:visible` decides whether an object is visible at all.
- `:stage` is identical to `:visible`, but when the object is not animated, it will be hidden by default.

26.4.1 Animating Color, Opacity, and Visibility

You can animate the color of the target object of an animation using the attributes `fill`, `draw`, and `text`. When the target of a color animation is a scope, you animate the color "used in this scope" for filling or stroking. However, when an object inside the scope has its color set explicitly, this color overrules the color of the scope.

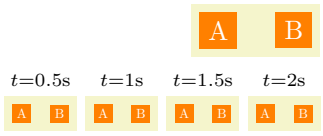
Animation attribute `:fill`, `:draw`




```
\usetikzlibrary {animations}
\tikz :fill = {0s = "red", 2s = "blue", begin on = click}
      [text = white, fill = orange ] {
  \node [fill]                    at (0mm,0) {A};
  \node [fill]                    at (5mm,0) {B};
  \node [fill = green!50!black ] at (1cm,0) {C};
}
```

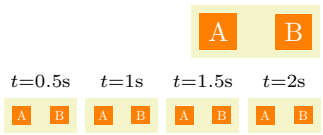
Animation attribute :text

The `text` attribute only applies to nodes and you need to directly animate the `text` attribute of each node individually.




```
\usetikzlibrary {animations}
\tikz [my anim/.style={ animate = {
          myself:text = {0s = "red", 2s = "blue", begin on = click}}},
       text = white, fill = orange ] {
  \node [fill, my anim] at (0,0) {A};
  \node [fill, my anim] at (1,0) {B};
}
```

Unlike the `fill` and `draw` colors, you cannot animate the `text` color for scopes:



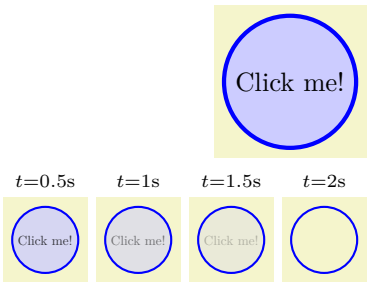

```
\usetikzlibrary {animations}
\tikz [animate = {myself:text = {0s = "red", 2s = "blue",
                                 begin on = click}},
       text = white, fill = orange ] {
  \node [fill] at (0,0) {A};
  \node [fill] at (1,0) {B};
}
```

Animation attribute :color

The `color` attribute is not really an attribute. Rather, it is a shorthand for `{draw,fill,text}`. This means that `color` does not start a separate timeline, but continues the `draw` timeline, the `fill` timeline, and the `text` timeline.

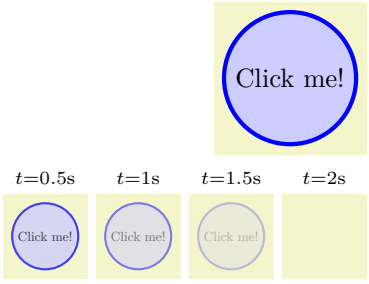
Animation attribute :opacity, :fill opacity, :stroke opacity

Similarly to the color, you can also set the opacity used for filling and for drawing using the attributes `fill opacity` and `draw opacity`, which are exactly the same as the usual TikZ keys of the same names.




```
\usetikzlibrary {animations}
\tikz \node :fill opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

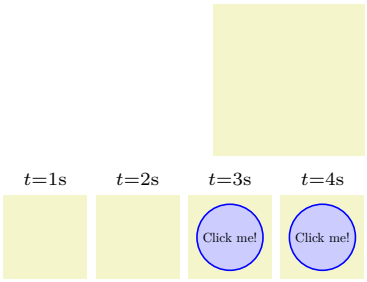
Unlike colors, where there is no joint attribute for filling and stroking, there is a single `opacity` attribute in addition to the above two attributes. If supported by the driver, it treats the graphic object to which it is applied as a transparency group. In essence, "this attribute does what you want" at least in most situations.




```
\usetikzlibrary {animations}
\tikz \node :opacity = { 0s="1", 2s="0", begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

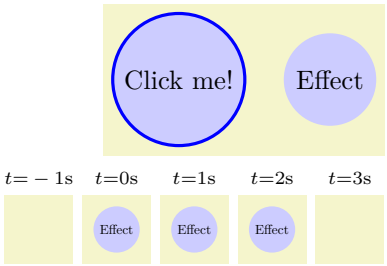
Animation attribute :visible, :stage

The difference between the `visible` attribute and an opacity of `0` is that an invisible object cannot be clicked and does not need to be rendered. The (only) two possible values for this attribute are `false` and `true`.




```
\usetikzlibrary {animations}
\tikz :visible = {begin on=click, 0s="false", 2s="false"}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
```

This `stage` attribute is the same as the `visible` attribute, only `base="false"` is set by default. This means that the object is *only* visible when you explicitly during the time the entries are set to `true`. The idea behind the name "stage" is that the object is normally "off stage" and when you explicitly set the "stage attribute" to `true` the object "enters" the stage and "leaves" once more when it is no longer "on stage".

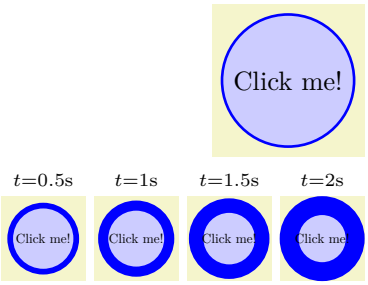



```
\usetikzlibrary {animations}
\tikz [animate = {example:stage = {
                    begin on = {click, of next=node},
                    0s="true", 2s="true" }}] {
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (2,0) (example) [fill = blue!20, circle] {Effect};
}
```

26.4.2 Animating Paths and their Rendering

The attributes of the appearance of a path that you can animate include the line width and the dash pattern, the path itself, as well as the arrow tips attached to the paths. Animating the line width and the dash pattern is easy since the animation attributes simply have that same names as the properties that they animate and the syntax for setting is also the same:

Animation attribute `:line width`

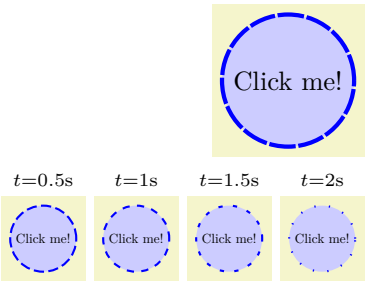



```
\usetikzlibrary {animations}
\tikz \node :line width = { 0s="1pt", 2s="5mm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

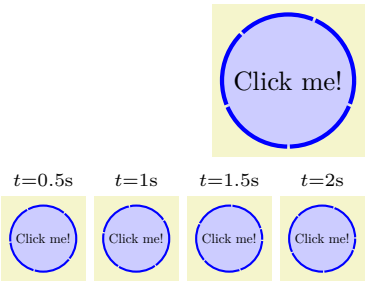
Note that you must specify number (or expressions that evaluate to numbers) as values, you cannot say `thin` or `thick` (these are styles, internally, and you also cannot say `line width=thick`).

Animation attribute `:dash`, `:dash phase`, `:dash phase`

The values for an animation of the dashing are specifications (see the `dash` key for details) consisting of a sequence of `on` and `off` numbers. In each value of the animation the length of these sequences *must* be identical. The interpolation of the values is done for each position of the sequences individually, and also on the phase.

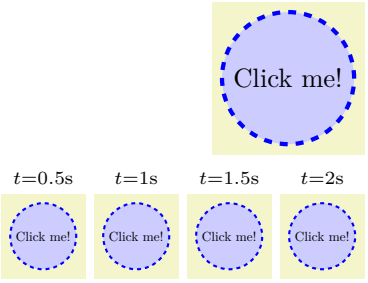



```
\usetikzlibrary {animations}
\tikz \node :dash = { 0s="on 10pt off 1pt  phase 0pt",
                      2s="on 1pt  off 10pt phase 0pt", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```




```
\usetikzlibrary {animations}
\tikz \node :dash = { 0s="on 1cm off 1pt phase 0pt",
                      2s="on 1cm off 1pt phase 1cm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

This `dash pattern` key allows you to animate the dash phase only. However, due to the way dashing is handled by certain drivers, the dash pattern is also set, namely to the current dash pattern that is in force when the animation is created.

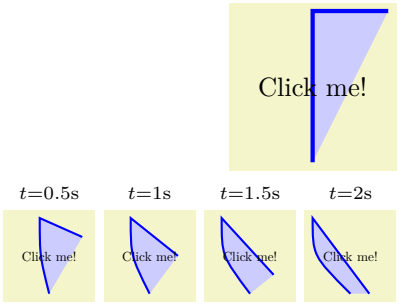



```
\usetikzlibrary {animations}
\tikz \node :dash phase = { 0s="0pt", 2s="1cm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle, dashed] {Click me!};
```

The above attributes "only" influence how the path is rendered. You can, however, also animate the path itself:

Animation attribute :path

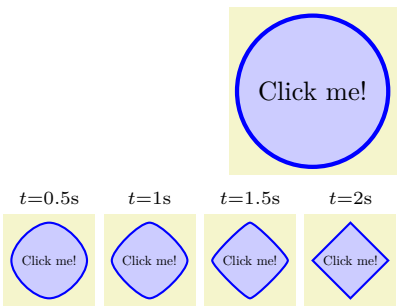
When you animate a path, the values are, of course, paths themselves:




```
\usetikzlibrary {animations}
\tikz \node :path = {
        0s = "{(0,-1) .. controls (0,0) and (0,0)    .. (0,1) -- (1,1)}",
        2s = "{(0,-1) .. controls (-1,0) and (-1,0) .. (-1,1) -- (.5,-1)}",
        begin on=click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

There a number of things to keep in mind when you animate a path:

- The path "values" are parsed and executed in an especially protected scope to ensure that they have only little side effects, but you should not do "fancy things" on these paths.
- As for the dash pattern, you must ensure that all paths in the timeline have the same structure (same sequence of path construction commands); only the coordinates may differ. In particular, you cannot say that the path at `1s` is a rectangle using `rectangle` and at `2s` is a circle using `circle`. Instead, you would have to ensure that at both times the path consists of appropriate Bézier curves (which is cumbersome as the following example shows, where we used the fact that a circle consists of four Bézier curves):




```
\usetikzlibrary {animations}
\tikz \node :path = {
        0s = "{(0,0) circle [radius=1cm]}",
        2s = "{(0,0)
               (1,0) .. controls +(0,0) and +(0,0) .. (0,1)
                     .. controls +(0,0) and +(0,0) .. (-1,0)
                     .. controls +(0,0) and +(0,0) .. (0,-1)
                     .. controls +(0,0) and +(0,0) .. (1,0)
                     -- cycle (0,0)}",
        begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

- You must specify arrow tips for an animated path in a special way, namely using the `arrows` key for *animations*, not the normal `arrows` key (see below).

`/tikz/animate/arrows=`⟨*arrow spec*⟩ (no default)

This key only has an effect on `:path` animations. It causes the arrow tips specified in ⟨*arrow spec*⟩ to be added to the path during the animation (the syntax is the same as for the normal `arrows` key). If you have several different animations for a paths, these may contain different arrow tips, but each animation must stick to one kind of arrow tips.

What happens internally when this key is used is the following: The specified arrow tips are rendered internally as so-called *markers,* which are small graphics that can be placed at the beginning and ends of paths and which "rotate along" as a path changes. Note that these markers are used *only* in the context of animated paths, the arrow tips of normal, "static" paths are drawn without the use of markers. Normally, there is no visual difference between an arrow tip drawn using markers or those drawn for static paths, but in rare cases there may be differences. You should only add arrows to open path consisting of a single segment with sufficiently long first and last segments (so that Ti*k*Z can shorten these segments correctly when necessary).

As pointed out earlier, the only way to add arrow tips to a path that is animated is using this key, you can *not* say something like

```
\draw :path = { 1s = "{(0,0) -- (1,0)}", 2s = "{(0,1) -- (1,0)}" }
  [->] (0,0) -- (1,0);
```

This will raise an error since you try to animate a path (`:path = ...`) that has normal arrow tips attached (`[->]`).

Instead, you must specify the arrow tips inside the animation command:

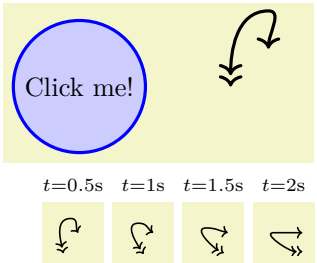
```
\draw :path = { 1s = "{(0,0) -- (1,0)}", 2s = "{(0,1) -- (1,0)}", arrows = -> }
  (0,0) -- (1,0);
```

However, the above code now has a big shortcoming: While the animation is *not* running, *no* arrow tip is shown (the `arrows` key only applies to the animation.

The trick is to use the `base` key. It allows you to install a path as the "base" path that is used when no animation is running and the arrows specified for the animation will also be used for the base. All told, the "correct" way to specify the animation is the following (note that no static path is specified, any specified path would be overruled by the `base` path anyway):

```
\draw :path = { 1s = "{(0,0) -- (1,0)}" base, 2s = "{(0,1) -- (1,0)}", arrows = -> };
```

Here is an example:




```
\usetikzlibrary {animations}
\tikz [very thick] {
  \node (node) at (-2,0)
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \draw :path = {
    0s = "{(0,0) to[out=90, in=180] (.5,1) to[out=0, in=90] (.5,.5)}" base,
    2s = "{(1,0) to[out=180, in=180] (.25,.5) to[out=0, in=180] (1,.5)}",
    arrows = <.<->, begin on = {click, of=node} }; }
```

`/tikz/animate/shorten <` = ⟨*dimension*⟩ (no default)

`/tikz/animate/shorten >` = ⟨*dimension*⟩ (no default)

For animated paths, just as the key `arrows` has to be passed to the animation (to `:path`) instead of to the static path, the keys `shorten >` and `shorten <` also have to be passed to the `:path` key.

26.4.3 Animating Transformations: Relative Transformations

In order to animate the canvas transformation matrix, you do not animate an attribute called "`:transform`". Rather, there are several attributes that all manipulate the canvas transformation matrix in different ways.

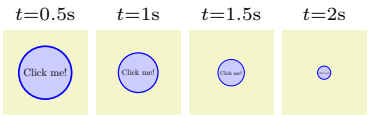
These keys, taken in appropriate combination, allow you to achieve any particular canvas transformation matrix. All keys that animate the transformation matrix always accumulate.

Let us start with the "standard" attributes that are also available as keys in Ti*k*Z:

Animation attribute :scale, :xscale, :yscale




```
\usetikzlibrary {animations}
\tikz \node :scale = { 0s="1", 2s="0.2", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

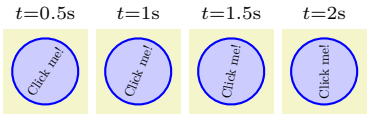


Animation attribute :rotate

The `rotate` key adds an animation of the rotation:




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="45", 2s="90", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



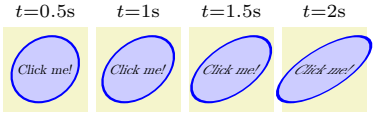
Note that there is no `rotate around` attribute, but you can use the `origin` key to change the origin of the rotation.

Animation attribute :xskew, :yskew, :xslant, :yslant

The keys add an animation of the skew (given in degrees) or slant (given as in the `xslant` and `yslant` key):

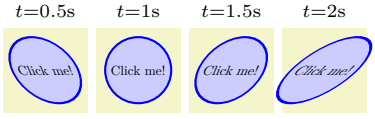



```
\usetikzlibrary {animations}
\tikz \node :xskew = { 0s="0", 2s="45", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```





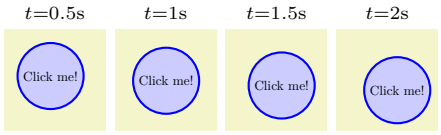

```
\usetikzlibrary {animations}
\tikz \node :xslant = { 0s="-1", 2s="1", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



Animation attribute :xshift, :yshift

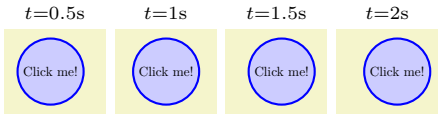



```
\usetikzlibrary {animations}
\tikz \node :shift = { 0s="{(0,0)}", 2s="{(5mm,-5mm)}",
                       begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```






```
\usetikzlibrary {animations}
\tikz \node :xshift = { 0s="0pt", 2s="5mm", begin on=click}
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

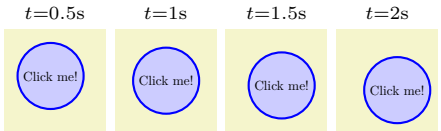


Animation attribute :shift

This `:shift` attribute can be animated in two ways. First, you can simply specify a sequence of coordinates in the same way as you would use the `shift` key in Ti*k*Z:



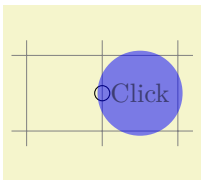

```
\usetikzlibrary {animations}
\tikz \node :shift = { 0s = "{(0,0)}", 2s = "{(5mm,-5mm)}",
                       begin on = click }
  [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



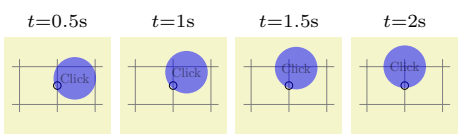
However, you can also specify the sequence of positions along which the shift should occur in a different way, namely by *specifying a path along which the object should be moved.* This is often not only more natural to do, but also allows you to specify movements along curves.

`/tikz/animate/options/along={`⟨*path*⟩`}`⟨`sloped` *or* `upright`⟩ `in`⟨*time*⟩ (no default)

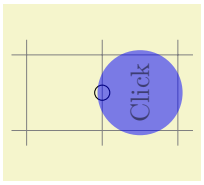
Use this key with a `:shift` (or a `:position`) to make Ti*k*Z shift the object by the coordinates along the ⟨*path*⟩. When this key is used, the values may no longer be coordinates, but must be fractions of the distance along the path. A value of `"0"` refers to the beginning of the path and `"1"` refers to the end:



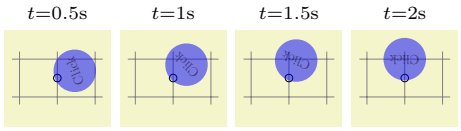

```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm];
  \node :shift = {
            along = {(0,0) circle[radius=5mm]} upright,
            0s="0", 2s=".25", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```



Following the ⟨*path*⟩, which must be put in braces, you must either specify `upright` or `sloped`. In the first case, the to-be-animated object is moved along the path normally (and stays "upright"), whereas when you use `sloped`, the object will be continuously rotated so that it always points along the path.

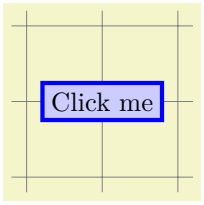



```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm];
  \node :shift = {
            along = {(0,0) circle[radius=5mm]} sloped,
            0s="0", 2s=".25", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

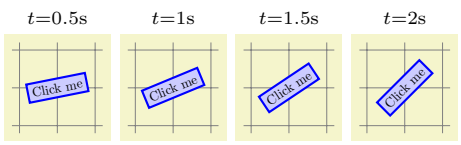


In most motion animations that use `along`, you will set the value for `0s` to `"0"` and the value for some specific ⟨*time*⟩ to `"1"`. Because of this, you can add `in` ⟨*time*⟩ after the path, to achieve exactly this effect.

For the above attributes, it is not immediately clear which coordinate system should be used for the animation. That is, when you move an object 1cm "to the right", where is "the right"? By default, movements and transformations like `:shift` or `:scale` are relative to the *animation coordinate system*, which defaults to the local coordinate system of the to-be-animated object. Consider the following example:

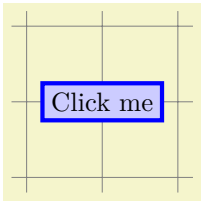



```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = { 0s="0", 2s="45", begin on=click}
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

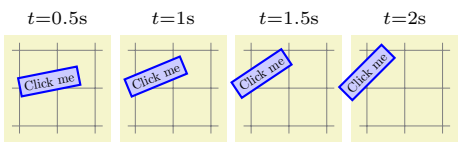


Note how the node rotates around its center even though this center is at position `(1,1)` in the picture's coordinate system. This is because `at (1,1)` actually only does a shift of the coordinate system and the node is then drawn at the origin of this shifted coordinate system. Since this shifted coordinate system becomes the animation coordinate system, the rotation "around the origin" is actually a rotation around the origin of the animation coordinate system, which is at `(1,1)` in the picture's coordinate system.

Let us, by comparison, do a rotation of a scope surrounding the node where the origin is not (yet) shifted:




```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \scoped :rotate = { 0s="0", 2s="45", begin on={click, of next=n} }
    \node (n) at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

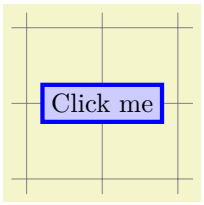


Now the rotation is really around the origin of the picture.

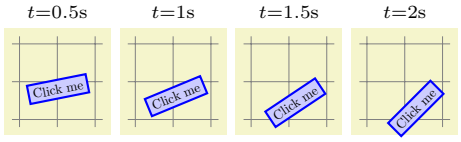
Most of the time the animation coordinate system will be setup in the way "you expect", but you can modify it using the following keys:

`/tikz/animate/options/origin=`⟨*coordinate*⟩ (no default)

Shifts the animation coordinate system by ⟨*coordinate*⟩. This has the effect that the "origin" for scalings and rotations gets shifted by this amount. In the following example, the point around which the rotation is done is the right border at `(2,1)` since the origin of the animation is at `(1,1)` relative to the picture's origin and the `origin` key shifts it one centimeter to the right.




```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :rotate = { 0s="0", 2s="45", begin on=click,
                    origin = {(1,0)}}
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```

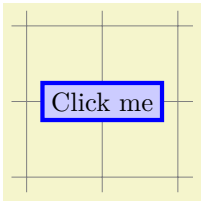


/tikz/animate/options/transform=⟨*transformation keys*⟩ (no default)

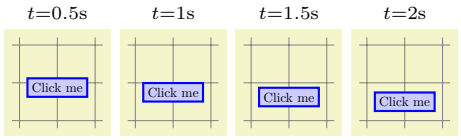
While the `origin` key does only a shift, the `transform` key allows you to add an arbitrary transformation to the animation coordinate system using keys like `shift`, `rotate` or even `reset cm` and `cm`. In particular, `origin=`⟨*c*⟩ has the same effect as `transform = {shift=`⟨*c*⟩`}`. Note that the transformation only influences the animation, not the object itself.

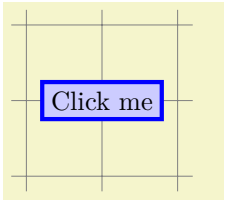
As an example, when you say `transform={scale=2}`, an `:xshift` with a value of `"1cm"` will actually shift the object by 2cm. Similarly, after you say `transform={rotate=90,scale=2}`, the same `:xshift` of `"1cm"` will actually shift the object by 2cm upwards.

Note that, internally, Ti*k*Z has to invert the transformation matrix resulting from the ⟨*transformation keys*⟩ (plus the original animation transformation matrix), which can by numerically instable when you use ill-conditioned transformations.

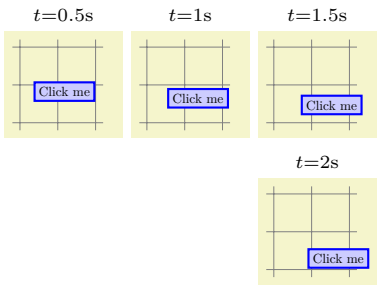



```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :xshift = { 0s="0cm", 2s="5mm", begin on=click,
                    transform = {rotate=-90} }
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```






```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,2.2);
  \node :xshift = { 0s="0cm", 2s="5mm", begin on=click,
                    transform = {rotate=-45, scale=2} }
    at (1,1) [fill = blue!20, draw = blue, ultra thick] {Click me};
}
```



26.4.4 Animating Transformations: Positioning

The attributes for specifying transformations and, in particular, the `:shift` attribute are always expressed in the local animation coordinate system. This makes it easy to "shift around a node a little bit", but makes it hard to move a node "from one position to another" since coordinates need to be expressed relative to the node's coordinate system, which leads to all sorts of problems: Suppose you wish to have a node move from

$(1,1)$ to $(2,1)$ and then to $(2,0)$. Now, if the node has already been placed at $(1,1)$ in the usual manner using `at`, then from the "node's point of view" you need to move the node to $(0,0)$, $(1,0)$, and $(1,-1)$. To make matters worse, when you use named coordinates as in

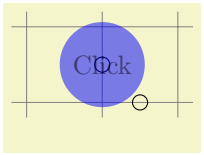
```
\coordinate(A) at (1,1);
\coordinate(B) at (2,1);
\coordinate(C) at (2,0);
```

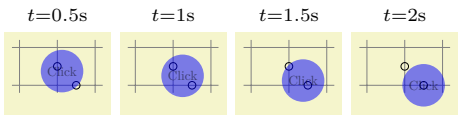
and then say that the movement should be from `(A)` to `(B)` to `(C)`, what should you expect? On the one hand, `(A)` and `(1,1)` should normally be interchangeable; on the other hand, `(A)` is a specific point in the plane, no matter from which coordinate system we look at it. It turns out that Ti*k*Z will stick to the second interpretation and actually turn `(A)` into `(0,0)` when it is parsed in the local coordinate system of a node starting at `(A)` – while `(1,1)` will stay the same.

Because of all these confusing effects, there is another attribute `:position`, which is similar to a `:shift`, but the coordinates are not interpreted in the local coordinate system of the node, but in the coordinate system that is in force when the `animate` key is used. For a node, this is *prior* to the setup of the node's coordinate system and, thus, usually the picture's coordinate system.

Animation attribute `:position`

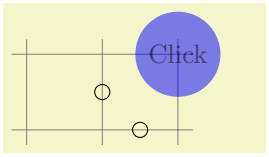
Compare the two animations, one with `:position`, one with `:shift`.

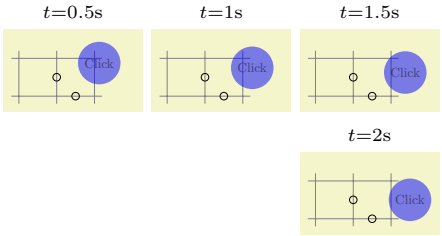





```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = { 0s="{(1,.5)}", 2s="{(1.5,0)}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

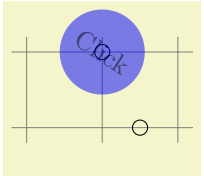
Compare this to a shift:

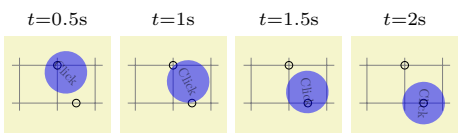





```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,.5) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :shift = { 0s="{(1,.5)}", 2s="{(1.5,0)}", begin on=click }
    at (1,.5) [fill = blue, opacity=.5, circle] {Click};
}
```

You can use the `along` key with `:position` in the same way as with `:shift`, which is especially useful for specifying that a node "travels" between positions of the canvas:






```
\usetikzlibrary {animations}
\tikz {
  \draw [help lines] (-0.2,-0.2) grid (2.2,1.2);
  \draw (1,1) circle [radius=1mm] (1.5,0) circle [radius=1mm];
  \node :position = {
            along = {(1,1) to[bend left] (1.5,0)} sloped in 2s,
            begin on = click }
          at (1,1) [fill = blue, opacity=.5, circle] {Click};
}
```

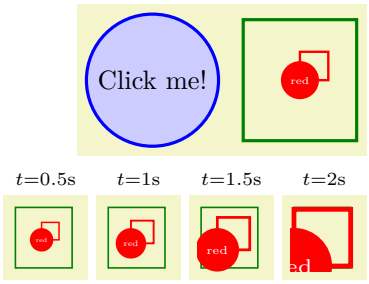
26.4.5 Animating Transformations: Views

The final method of changing the transformation matrix is to animate a *view*.

Animation attribute :view

A view is a canvas transformation that shifts and scales the canvas in such a way that a certain rectangle "matches" another rectangle: The idea is that you "look through" a "window" (the view) and "see" a certain area of the canvas. View animation do not allow you to do anything that cannot also be done using the `shift` and `scale` keys in combination, but it often much more natural to animate which area of a graphic you wish to see than to compute and animate a scaling and shift explicitly.

In order to use a view, you first need to create a view, which is done using the `meet` or `slice` keys from the `views` library, see Section 78. You can then animate the view using the `view` attribute. The values passed to the `entry` key follow the same syntax as the views in the `views` library (though you only animate the to-be-viewed rectangle).




```
\usetikzlibrary {animations,views}
\tikz [very thick] {
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};

  \draw [green!50!black] (1.2,-0.8) rectangle (2.7,0.8);
  \begin{scope}[view = {(0,0) (2,2) at (1.2,-0.8) (2.7,0.8)},
                animate = {myself:view = {
                    begin on = {click, of=node},
                    0s = "{(0,0) (2,2)}",
                    2s = "{(1,1) (1.5,1.5)}" }}]
    \draw [red] (10mm,10mm) rectangle (15mm,15mm);
    \node at (10mm,10mm) [circle, fill=red, text=white, font=\tiny] {red};
  \end{scope}
}
```

26.5 Controlling the Timeline

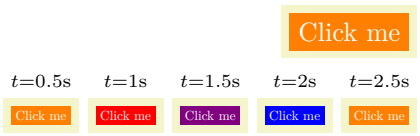
We can already specify timelines by giving a sequence of times in non-decreasing order along with corresponding values. In this section we have a look at further options that allow us to extend or control the timeline.

26.5.1 Before and After the Timeline: Value Filling

When you specify the timeline, you specify it for a certain interval $[t_1, t_2]$. By default, outside this interval the animation has no effect on the to-be-animated attribute. The following keys allows you to change this:

`/tikz/animate/base`=⟨*options*⟩ (no default)

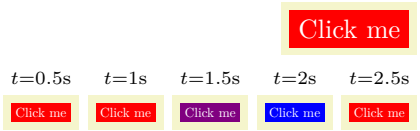
A "base" value is a value that is used for the attribute whenever the timeline is *not* active:




```
\usetikzlibrary {animations}
\tikz \node [fill = green, text = white] :fill =
    { 1s = "red", 2s = "blue", base = "orange", begin on = click }
  {Click me};
```

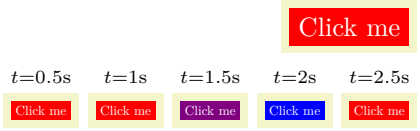
Syntactically, the `base` key works much like special time syntax: It sets up a local `sync` scope and executes the ⟨*options*⟩ in it and creates an `entry`. However, instead of setting the `time` attribute to a time, it sets it to a special value that tells TikZ that when the entry is created, the current ⟨*value*⟩ should be used as the `base` value.

This means that you can write `base = "orange"` as in the above example to set the base. However, you can also use the `base` key in other ways; most noticeably, you can use it *after* some value:




```
\usetikzlibrary {animations}
\tikz \node [fill = green, text = white] :fill =
    { 1s = {"red" = base}, 2s = "blue", begin on = click }
  {Click me};
```

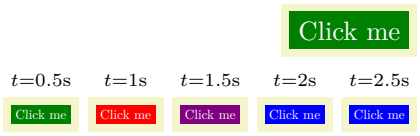
Instead of using `base` as a key, you can also add `base` directly after the quotes of a value. This is particularly useful for setting up a base value that is also used in a timeline:



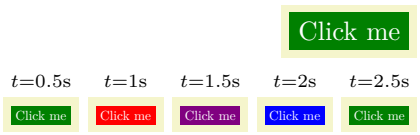

```
\usetikzlibrary {animations}
\tikz \node [fill = green, text = white] :fill =
    { 1s = "red" base, 2s = "blue", begin on = click }
  {Click me};
```

/tikz/animate/options/forever (no value)

This key causes the timeline to continue "forever" after the last time with the last value. You can also think of this as having the animation "freeze" at the end.




```
\usetikzlibrary {animations}
\tikz \node :fill = { 1s="red", 2s="blue", forever, begin on=click}
  [fill = green!50!black, text = white] {Click me};
```




```
\usetikzlibrary {animations}
\tikz \node [fill = green!50!black, text = white]
    :fill = { 1s = "red", 2s = "blue", begin on = click }
  {Click me};
```

/tikz/animate/options/freeze (no value)

An alias for `forever`.

26.5.2 Beginning and Ending Timelines

The ⟨*time*⟩ used with the first use of the `entry` key in a timeline is the start time and the ⟨*time*⟩ in the last `entry` key is the stop time. However, this leaves open then question of when the whole timeline is to be started: The moment the document is opened? When the page is displayed? When the user scrolls to the to-be-animated object? When some other object is clicked? The key `begin`, and also the key `end`, allow you to specify answers to these questions.

/tikz/animate/options/begin=⟨*time*⟩ (no default)

This key specifies when the "moment `0s`" should be relative to the moment when the current graphic is first displayed. You can use this key multiple times, in this case the timeline is restarted for each of the times specified (if it is already running, it will be reset). If no `begin` key is given at all, the effect is the same as if `begin=0s` had been specified.

It is permissible to set ⟨*time*⟩ to a negative value.

Note that this key has no effect for snapshots.

/tikz/animate/options/end=⟨*time*⟩ (no default)

This key will truncate the timeline so that it ends ⟨*time*⟩ after the display of the graphic, provided the timeline begins before the specified end time. For instance, if you specify a timeline starting at 2 s and ending at 5 s and you set `begin` to 1 s and `end` to 4 s, the timeline will run, relative to the moment when the graphic is displayed from 3 s to 4 s.




```
\usetikzlibrary {animations}
\tikz \node [fill = green!50!black, text = white]
    :rotate = { 1s = "0", 5s = "90", begin = 2s, end = 4s }
  {Click me};
```

Instead of specifying the beginning of the timeline relative to the moment to to-be-animated graphic is displayed, you can also set the "moment `0s`" to the moment a specific *event* happens using the following key:

/tikz/animate/options/begin on=⟨*options*⟩ (no default)

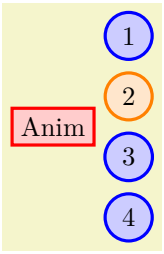
The ⟨*options*⟩ will be executed with the path `/pgf/animation/events` and will cause a new beginning to be added to the list of possible beginnings for the timeline (so the uses of this key accumulate). Each "beginning" is just another possible "moment `0s`" for the timeline. For instance, when the ⟨*options*⟩ are set to `click`, then each time the graph is clicked a moment `0s` starts for the timeline.

Most events are "caused" or "happen to" some object. For instance, the `click` event happens when you click on a certain object. In order to specify this object, use the following two keys inside the ⟨*options*⟩: `of` and `of next`. If neither of these keys are given, the to-be-animated object is used.

/pgf/animation/events/of=⟨*id*⟩.⟨*type*⟩ (no default)

This specifies a graphic object id in the same way as the `whom` key, also with an optional ⟨*type*⟩. This is the object that "causes" the event to happen.

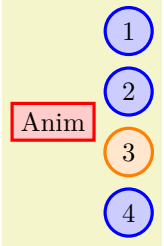
Unlike the `whom` key, which always refers to a not-yet-existing object, this key always refers to an already existing object, namely to the most recent use of the ⟨*id*⟩. In the following example, the referenced object is the node with the label `2` since it is the most recently referenced node with ⟨*id*⟩ `X`.




```
\usetikzlibrary {animations}
\tikz [very thick] {
  \node (X) at (1,1.2)  [fill = blue!20, draw = blue, circle] {1};
  \node (X) at (1,0.4)  [fill = orange!20, draw = orange, circle] {2};
  \node (node) :rotate = {0s="0", 2s="90", begin on = {click, of = X}}
               [fill = red!20, draw = red, rectangle] {Anim};
  \node (X) at (1,-0.4) [fill = blue!20, draw = blue, circle] {3};
  \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4}; }
```

/pgf/animation/events/of next=⟨*id*⟩.⟨*type*⟩ (no default)

This key works like the `of` key, only it refers to a future (actually, the next) object with the given ⟨*id*⟩, not to a previous one. This, in the next example, the referenced node is the one with label `3`.




```
\usetikzlibrary {animations}
\tikz [very thick] {
  \node (X) at (1,1.2)  [fill = blue!20, draw = blue, circle] {1};
  \node (X) at (1,0.4)  [fill = blue!20, draw = blue, circle] {2};
  \node (node) :rotate = {
                 0s="0", 2s="90", begin on = {click, of next = X}}
               [fill = red!20, draw = red, rectangle] {Anim};
  \node (X) at (1,-0.4) [fill = orange!20, draw = orange, circle] {3};
  \node (X) at (1,-1.2) [fill = blue!20, draw = blue, circle] {4}; }
```

The following key allows you to specify the event that should cause the animation to start:

/pgf/animation/events/event=⟨*event name*⟩ (no default)

Specifies the name of the event whose occurrence should start the timeline. Which events are supported depends on the device on which the animation is displayed, the output format (SVG or some other format), and the setup of scripts, but here is a list of events supported by "plain SVG": `click`, `focusin`, `focusout`, `mousedown`, `mouseup`, `mouseover`, `mousemove`, `mouseout`, `begin`, `end`. However, the following keys make using these events simpler:

/pgf/animate/events/click (no value)

This is a shorthand for `event=click`. This event gets triggered when the user clicks on the triggering object with a mouse (or something equivalent).




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {click}}
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse down (no value)

Shorthand for `event=mousedown`. The event gets triggered when the user presses a mouse button down on the object.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse down}}
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse up (no value)

Shorthand for `event=mouseup` and gets triggered, of course, when a pressed button is released on the object.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse up} }
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse over (no value)

Shorthand for `event=mouseover`. The event gets triggered the moment the mouse cursor moves over the object.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse over} }
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse move (no value)

Shorthand for `event=mousemove`. The event gets triggered lots of times, namely each time the mouse moves while being "over" the object.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse move} }
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/mouse out (no value)

Shorthand for `event=mouseout`. The opposite of `mouse over`: triggered when the mouse leaves the object.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90", begin on = {mouse out} }
  [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/begin (no value)

Shorthand for `event=begin`. The "begin" refers to the beginning of another animation, namely the one referenced by `of` or `of whom`. This means that the current animation will begin when some other animation begins.




```
\usetikzlibrary {animations}
\tikz \node [animate = {
      myself:rotate = { 0s="0", 2s="90", begin on = {begin, of next=anim}},
      myself:xshift = { 0s="0mm", 2s="5mm", begin on = {click}, name=anim}
    },
    fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/end (no value)

Shorthand for `event=end`. Again, the "end" refers to the end of another animation, namely the one referenced by `of` or `of whom`. This means that the current animation will *begin* when some other animation *ends*.




```
\usetikzlibrary {animations}
\tikz \node [animate = {
    myself:rotate = { 0s="0", 2s="90", begin on = {end, of next=anim}},
    myself:xshift = { 0s="0mm", 2s="5mm", begin on = {click}, name=anim }
  },
  fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/focus in (no value)

This is a shorthand for `event=focusin`. This event gets triggered when the graphic object gets the focus (this usually makes sense only for text input fields).

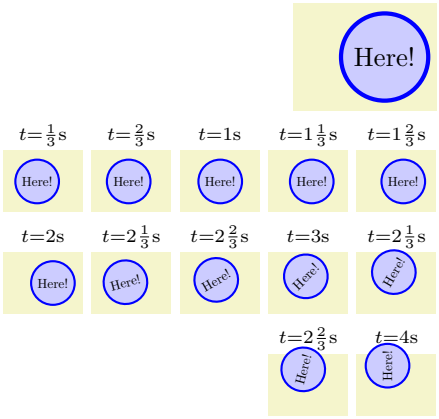
/pgf/animation/events/focus out (no value)

This is a shorthand for `event=focusout`.

In addition to the events specified using the generic `event` key, there are two further events that take a parameter:

/pgf/animation/events/repeat=⟨*number*⟩ (no default)

The event is triggered when a repeating animation has been repeated ⟨*number*⟩ times.




```
\usetikzlibrary {animations}
\tikz
  \node [animate = { myself: = {
    :rotate = { 0s="0", 2s="90", begin on = {repeat = 2, of next = anim },
                begin snapshot = 2 },
    :xshift = { 0s="0mm", 2s="5mm", begin on=click, name=anim, repeats=4 }}},
    fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

/pgf/animation/events/key=⟨*key*⟩ (no default)

The event is triggered when the keyboard key ⟨*key*⟩ has been pressed. For security reasons, a viewer may suppress this.

Having specified the event, you can also specify a delay relative to this event:

/pgf/animation/events/delay=⟨*time*⟩ (no default)

Specifies that the timeline should not start with the event, but, rather, be delayed by ⟨*time*⟩.

When you use `begin on` to start an animation when a certain event is triggered, it is not clear what should happen when the event is triggered *again*. Should this be ignored completely? Should it only be ignored while the animation is running? The following key allows you to specify when should happen:

/tikz/animate/options/restart=⟨*choice*⟩ (default `true`)

You can set ⟨*choice*⟩ to one of the following:

- `true` means that the animation will restart each time the event is triggered. If the animation is already running, it will be reset to its beginning.
- `false` means that once the animation has started once, it will never be restarted.

Here!

```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90",
                        restart = false, begin on = {click}}
    [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

- `never` means the same as `false`.
- `when not active` means that the animation will restart when the event is triggered, but *not* while the animation is running.

Here!

```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s="0", 2s="90",
                        restart = when not active, begin on = {click}}
    [fill = blue!20, draw = blue, circle, ultra thick] {Here!};
```

Just like `begin on` specifies when a timeline begins relative to some event, the `end on` allows you to stop is early when some event happens:

`/tikz/animate/options/end on`=⟨*options*⟩ (no default)

Works exactly like `begin on`, one possible end of the timeline is specified using the ⟨*options*⟩.

26.5.3 Repeating Timelines and Accumulation

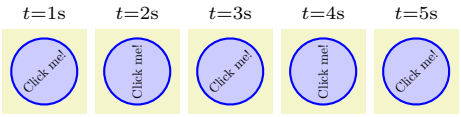
`/tikz/animate/options/repeats`=⟨*specification*⟩ (no default)

Use this key to specify that the timeline animation should repeat at the end. The ⟨*specification*⟩ must consist of two parts, each of which may be empty. The first part is one of the following:

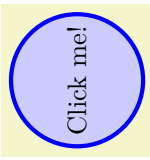
- Empty, in which case the timeline repeats forever.



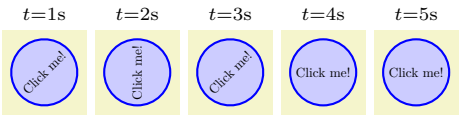

```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats, begin on = click }
    [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



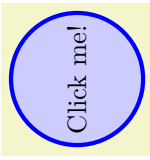
- A ⟨*number*⟩ (like `2` or `3.25`), in which case the timeline repeats ⟨*number*⟩ times.



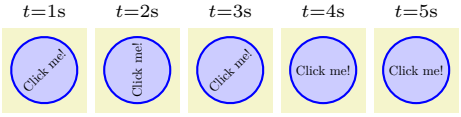

```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats = 1.75, begin on = click }
    [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```



- The text "`for` ⟨*time*⟩" (like `for 2s` or `for 300ms`), in which case the timeline repeats however often necessary so that it stops exactly after ⟨*time*⟩.

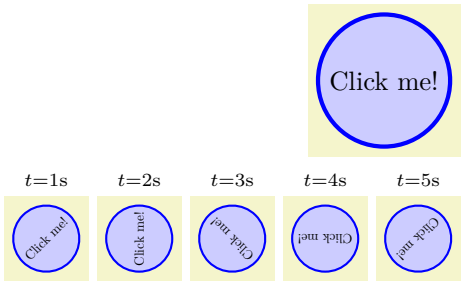



```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90",
                        repeats = for 3.5s, begin on = click }
    [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

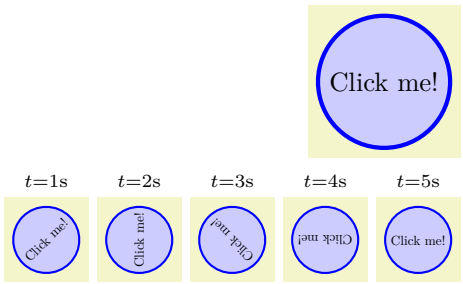


The second part of the specification must be one of the following:

- Empty, in which case each time the timeline is restarted, the attribute's value undergoes the same series of values it did previously.
- The text `accumulating`. This has the effect that each time the timeline is restarted, the last values specified by the timeline is *added* to the value from the previous iteration(s). A typical example is an animation that shifts a scope by, say, 1 cm over a time of 1 s. Now, if you repeat this five times, normally the scope will shift 1 cm for 1 s then "jump back", shift again, jump back, and so on for five times. In contrast, when the repeats are accumulating, the scope will move by 5 cm over 5 s in total.




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90", begin on = click,
                        repeats = accumulating }
    [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```




```
\usetikzlibrary {animations}
\tikz \node :rotate = { 0s = "0", 2s = "90", begin on = click,
                        repeats = for 4s accumulating }
    [fill = blue!20, draw = blue, ultra thick, circle] {Click me!};
```

/tikz/animate/options/repeat=⟨*specification*⟩ (no default)

An alias for `repeats`.

26.5.4 Smoothing and Jumping Timelines

Your specification of the timeline will consist of a sequence of times along with values that the attribute should have at these "key times". Between these key times, the attribute's value needs to be interpolated.

Suppose that an animation is supposed to interpolate a attribute's value between the two values `50` and `100` over a time of 10 s. The simplest way of doing so is to do a linear interpolation, where the value as, say, 1 s is 55, at 2 s it is 60, and so on. Unfortunately, the linear interpolation does not "look" nice in many cases since the acceleration of a linear interpolation is zero during the animation, but infinite at the beginning and at the end; which looks "jerky".

To avoid this, you can specify that the time–attribute curve should not be a straight line, but rather a curve. You specify this curve using a spline. The most logical "coordinate rectangle" used for this spline in our example would be `(0s,50)` and `(10s,100)` and we would like to specify something like

```
(0s,50) .. controls (5s,50) and (9s,100) .. (10s,100)
```

This would result in a time–attribute curve where the attribute at `50` changes slowly at 0 s and also arrives slowly at `100` at 10 s, but speeds up between these values.

We call the first control point `(5s,50)` the "exit control" and call `(9s,100)` the "entry control": The first control dictates how quickly or slowly a time point is left, the second dictates how quickly or slowly we enter the next one.

The control points are, however, not specified in the coordinate system indicated above. Rather, the rectangle `(0s,50)` to `(10s, 100)` gets normalized to `(0,0)` to `(1,1)`. The control point `(5s,50)` would thus become `(0.5,0)` and `(9s,100)` becomes `(0.9,1)`.

/tikz/animate/options/exit control={⟨*time fraction*⟩}{⟨*value fraction*⟩} (no default)

Specifies an exit control using two values as above. The spline from above would be specified as follows:

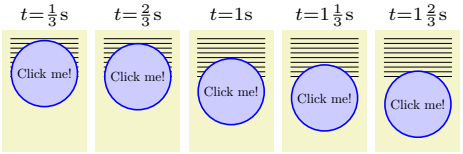
```
exit control={0.5}{0},
entry control={0.9}{1},
0s = "50",
10s = "100"
```

Note that the curve specified using exit and entry controls must be "well-behaved" in the sense that exactly one value must be specified for each point in time in the time interval.

In the next three example, we first specify a "smooth" exit from the start position, then a smooth arrival at the end position, and, finally both.

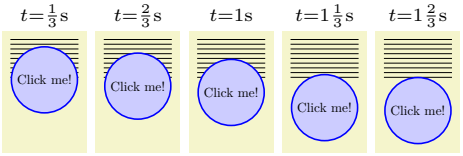



```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click,
                    0s = { exit control = {1}{0}, "0cm" },
                    1s = "-5mm",
                    2s = "-10mm" }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```



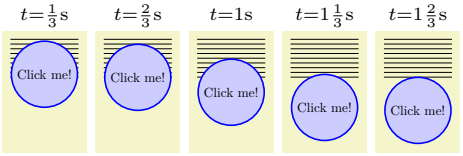



```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click,
                    0s = "0cm",
                    1s = "-5mm",
                    2s = { entry control = {0}{1}, "-10mm" } }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```






```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click,
                    0s = { exit control = {1}{0}, "0cm" },
                    1s = "-5mm",
                    2s = { entry control = {0}{1}, "-10mm" } }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```



`/tikz/animate/options/entry control={`⟨*time fraction*⟩`}{`⟨*value fraction*⟩`}` (no default)

Works like `exit control`.

`/tikz/animate/options/ease in={`⟨*fraction*⟩`}` (default `0.5`)

A shorthand for `entry control={1-`⟨*fraction*⟩`}{1}`.

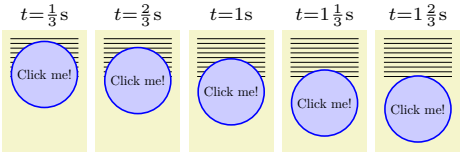
`/tikz/animate/options/ease out={`⟨*fraction*⟩`}` (default `0.5`)

A shorthand for `exit control={`⟨*fraction*⟩`}{1}`.

`/tikz/animate/options/ease={`⟨*fraction*⟩`}` (default `0.5`)

A shorthand for `ease in=`⟨*fraction*⟩`, ease out=`⟨*fraction*⟩.

Note that since for the first time the entry control is ignored and, similarly, for the last time the exit control is ignored, using the `ease` key with an animation having only two times is particularly easy, since we only need to set `ease` once:

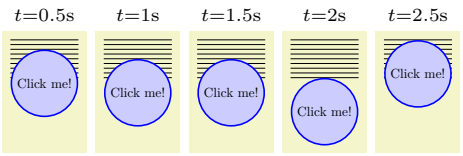



```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click, ease, 0s = "0cm", 2s = "-10mm" }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

The opposite of having a smooth curve between two values, is to have a "jump" from one value to the next. There are two keys for this:

`/tikz/animate/options/stay` (no value)

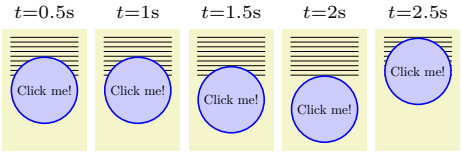
Specifies that inside the time interval the value "stays put" at the first value till the end of the interval, where it will jump to the second value. This is similar to an exit control where the curve is "infinitely flat".




```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click,
                    0s = "0cm",
                    1s = {stay, "-5mm"},
                    2s = "-10mm" }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`/tikz/animate/options/jump` (no value)

Works like the `stay` key, but will cause the value to "jump to" the new value right at the beginning of the time interval. It is similar to an entry control specifying a "flat" curve.



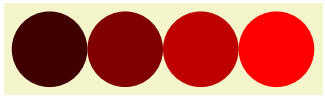

```
\usetikzlibrary {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \node :yshift = { begin on = click,
                    0s = "0cm",
                    1s = {jump, "-5mm"},
                    2s = "-10mm" }
    [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

26.6 Snapshots

Snapshots are a way of taking a "photographic snapshot" of an animation at a certain time and then insert these into PDF files (or, for that matter, Postscript files or files in any other format, including SVG): You specify a time like `2s` and then Ti*k*Z will compute what the animation "would look like after 2s" and insert the necessary graphics command for rendering the graphic objects in the correct way. Since this computation

is done by Ti*k*Z and since only "normal" graphics command are inserted into the output, snapshots work with all output formats.

Apart from providing a fallback for PDF, snapshots are very useful by themselves: They make it easy to "show" how an animation unfolds on paper. For this, you simply typeset the same picture with the same animation several times (using a simple `\foreach` loop), but each time you set a different snapshot time. This will result in a sequence of pictures that depict the animation at different points in time and which can then be inserted alongside each other into the printed document. This approach has been used with the examples of animations in this manual.



```
\usetikzlibrary {animations}
\foreach \t in {0.5, 1, 1.5, 2}
  \tikz [make snapshot of = \t]
    \fill :fill = {0s="black", 2s="red"} (0,0) circle [radius = 5mm];
```

Creating snapshots is done using the following key:

`/tikz/make snapshot of`=⟨*time*⟩ (no default)

When this key is used in a TEX scope, animation commands given in the scope do not add animation code to the output. Instead, Ti*k*Z computes the values the attributes of the animation would have at the specified ⟨*time*⟩ and inserts the necessary system layer command to set the attribute to the computed values (some care has been taken to make this computation match the computations done by viewer applications as best as possible).



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white" } (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle ++(1,1);
}
```

The moment ⟨*time*⟩ is best thought of as ⟨*time*⟩ seconds after the "moment zero" where all timelines start by default. Now, "real" animation may start at different time through user interaction, which clearly makes no sense for snapshots. Nevertheless, you will sometimes wish to have more control over when a timeline starts for the purposes of taking snapshots. You can use the following key for this:

`/tikz/animate/options/begin snapshot`=⟨*start time*⟩ (no default)

Use this key on a timeline to specify that, only for purposes of taking snapshots, the timeline starts at ⟨*start time*⟩ rather than at "moment zero". (Think of this as saying that the animation starts when a virtual user clicks on the animation and this click occurs ⟨*start time*⟩ seconds after the general "moment zero", causing the animation to "lag behind" by this amount of time.) Computationally, for the timeline the ⟨*start time*⟩ is subtracted from the snapshot's ⟨*time*⟩ when the value needs to be determined:



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 1s] {
  \fill :fill = { 0s = "black", 2s = "white",
                  begin snapshot = 1s }        (0,0) rectangle ++(1,1);
  \fill :fill = { 1s = "black", 3s = "white" } (2,0) rectangle ++(1,1);
}
```

The computations of the values the animation "would have" are done entirely by Ti*k*Z, which has the big advantage is that no support from the viewer application or the output format is needed – snapshots work with all output formats, not just with SVG. However, computations done by Ti*k*Z are not always very precise and can be slow because of TEX's limitations. In addition, there are some further limitations when it comes to Ti*k*Z's computation of snapshot values:

- As mentioned above, except for `begin snapshot`, other commands for specifying the beginning or end of a timeline based on user interaction make no sense for timelines: The keys `begin`, `begin on`, `end`, and `end on` are silently ignored.
- The value `current value` for a value is forbidden since this value is almost impossible to compute by Ti*k*Z.
- Accumulating repeats of a motion are (currently) not supported, but should not rely on this.

When ⟨*time*⟩ is empty, "snapshot taking" is switched off and animation commands are inserted once more.

`/tikz/make snapshot after`=⟨*time*⟩ (no default)

Works exactly like `make snapshot of`, only the ⟨*time*⟩ is interpreted as ⟨*time*⟩ + ϵ. This only makes a difference at the end of a timeline and when there are two or more values specified for the same time: When there are several values specified for time t, a normal snapshot for time t uses the first value given for the attribute. In contrast, this command would use the last one given. Similarly, when an animation timeline ends at time t, a normal snapshot of time t would use the last value of the timeline, while this key would not apply the animation at all (it has already ended at time $t + \epsilon$).



```
\usetikzlibrary {animations}
\tikz [make snapshot of = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
\tikz [make snapshot after = 2s]
  \fill :fill = { 0s = "green", 2s = "red" } (0,0) rectangle ++(1,1);
```

`/tikz/make snapshot if necessary`=⟨*time*⟩ (default `0s`)

This key makes a snapshot of ⟨*time*⟩ only when the output format does not provide support for animations; if the output format supports animations (like SVG), then the command has no effect and animations are created normally.

This manual is typeset with the following being set once are for all in preamble:

```
\tikzset{make snapshot if necessary}
```

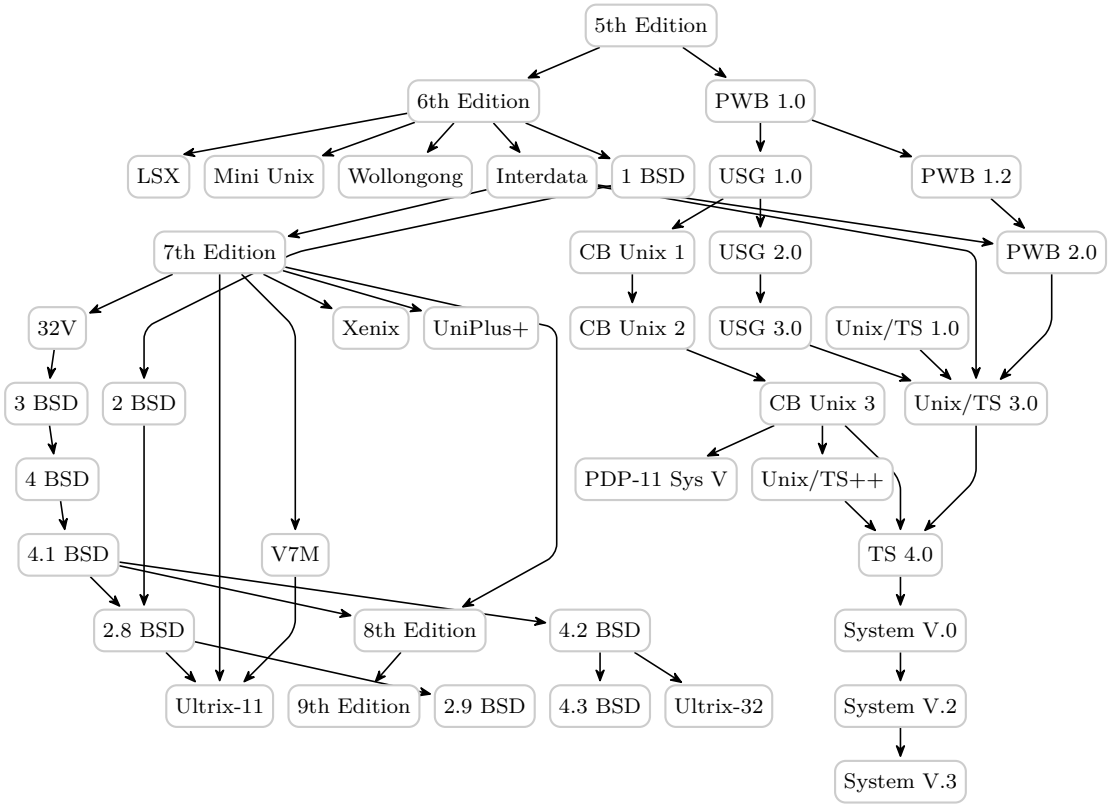
Because of this setting, in the PDF version of this document, all animations are shown at the value they would have at moment 0*s*. In contrast, in the SVG version, the animations are created normally.

In both versions, the smaller pictures showing how the animation proceeds over time are created using `make snapshot of` for the indicated times.

Part IV
Graph Drawing

by Till Tantau et al.

Graph drawing algorithms do the tough work of computing a layout of a graph for you. Ti*k*Z comes with powerful such algorithms, but you can also implement new algorithms in the Lua programming language.



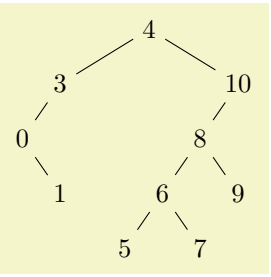

```
\usetikzlibrary {arrows.meta,graphs,graphdrawing} \usegdlibrary {layered}
\tikz [nodes={text height=.7em, text depth=.2em,
              draw=black!20, thick, fill=white, font=\footnotesize},
       >={Stealth[round,sep]}, rounded corners, semithick]
  \graph [layered layout, level distance=1cm, sibling sep=.5em, sibling distance=1cm] {
    "5th Edition" -> { "6th Edition", "PWB 1.0" };
    "6th Edition" -> { "LSX" [>child anchor=45],  "1 BSD", "Mini Unix", "Wollongong", "Interdata" };
    "Interdata" -> { "Unix/TS 3.0", "PWB 2.0", "7th Edition" };
    "7th Edition" -> { "8th Edition", "32V", "V7M", "Ultrix-11", "Xenix", "UniPlus+" };
    "V7M" -> "Ultrix-11";
    "8th Edition" -> "9th Edition";
    "1 BSD" -> "2 BSD" -> "2.8 BSD" -> { "Ultrix-11", "2.9 BSD" };
    "32V" -> "3 BSD" -> "4 BSD" -> "4.1 BSD" -> { "4.2 BSD", "2.8 BSD", "8th Edition" };
    "4.2 BSD" -> { "4.3 BSD", "Ultrix-32" };
    "PWB 1.0" -> { "PWB 1.2" -> "PWB 2.0", "USG 1.0" -> { "CB Unix 1", "USG 2.0" }};
    "CB Unix 1" -> "CB Unix 2" -> "CB Unix 3" -> { "Unix/TS++", "PDP-11 Sys V" };
    { "USG 2.0" -> "USG 3.0", "PWB 2.0", "Unix/TS 1.0" } -> "Unix/TS 3.0";
    { "Unix/TS++", "CB Unix 3", "Unix/TS 3.0" } -> "TS 4.0" -> "System V.0" -> "System V.2" -> "System V.3";
  };
```

27 Introduction to Algorithmic Graph Drawing

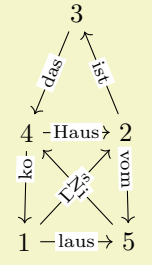
by Till Tantau

27.1 What Is Algorithmic Graph Drawing?

Algorithmic graph drawing (or just *graph drawing* in the following) is the process of computing algorithmically where the nodes of a graph are positioned on a page so that the graph "looks nice". The idea is that you, as human (or you, as a machine, if you happen to be a machine and happen to be reading this document) just specify which nodes are present in a graph and which edges are present. Additionally, you may add some "hints" like "this node should be near the center" or "this edge is pretty important". You do *not* specify where, exactly, the nodes and edges should be. This is something you leave to a *graph drawing algorithm*. The algorithm gets your description of the graph as an input and then decides where the nodes should go on the page.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm] {
  4 -- {
    3 -- 0 -- 1[second],
    10 -- {
      8 -- {
        6 -- {5,7},
        9
  } } }
};
```



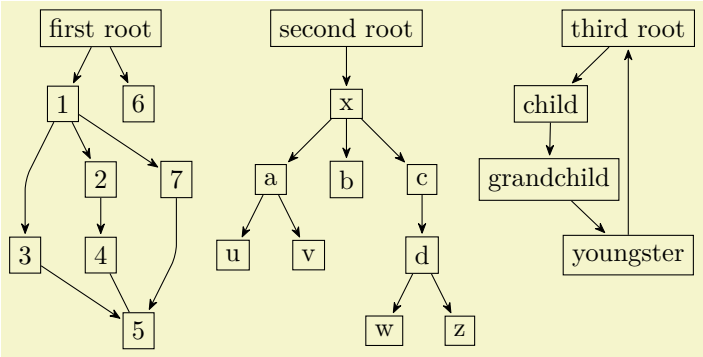

```
\usetikzlibrary {graphs,graphdrawing,quotes} \usegdlibrary {force}
\tikz \graph [spring layout,
  edge quotes mid,
  edges={nodes={font=\scriptsize, fill=white, sloped, inner sep=1pt}}]
{
  1 ->["Das"] 2 ->["ist"] 3 ->["das"] 4 ->["Haus"]
  2 ->["vom" near start] 5 ->["Ni"] 4 ->["ko" near start]
  1 ->["laus", orient=right] 5;
};
```

Naturally, graph drawing is a bit of a (black?) art. There is no "perfect" way of drawing a graph, rather, depending on the circumstances there are several different ways of drawing the same graph and often it will just depend on the aesthetic sense of the reader which layout he or she would prefer. For this reason, there are a huge number of graph drawing algorithms "out there" and there are scientific conference devoted to such algorithms, where each year dozens of new algorithms are proposed.

Unlike the rest of PGF and Ti*k*Z, which is implemented purely in TEX, the graph drawing algorithms are simply too complex to be implemented directly in TEX. Instead, the programming language Lua is used by the `graphdrawing` library – a programming language that has been integrated into recent versions of TEX. This means that (a) as a user of the graph drawing engine you run TEX on your documents in the usual way, no external programs are called since Lua is already integrated into TEX, and (b) it is pretty easy to implement new graph drawing algorithms for Ti*k*Z since Lua can be used and no TEX programming knowledge is needed.

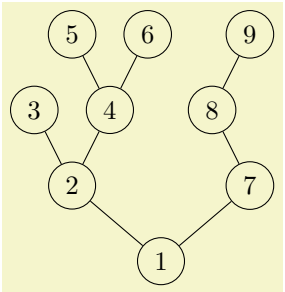
27.2 Using the Graph Drawing System

"Users" of the graph drawing engine can invoke the graph drawing algorithms often by just adding a single option to their picture. Here is a typical example, where the `layered layout` option tells Ti*k*Z that the graph should be drawn ("should be laid out") using a so-called "layered graph drawing algorithm" (what these are will be explained later):



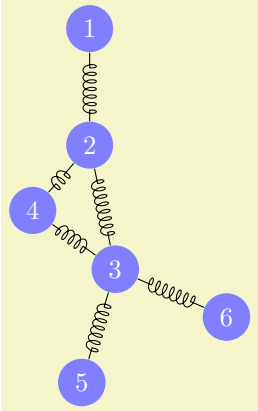

```
\usetikzlibrary {arrows.meta,graphs,graphdrawing} \usegdlibrary {layered}
\tikz [>={Stealth[round,sep]}]
  \graph [layered layout, components go right top aligned, nodes=draw, edges=rounded corners]
  {
    first root -> {1 -> {2, 3, 7} -> {4, 5}, 6 }, 4 -- 5;
    second root -> x -> {a -> {u,v}, b, c -> d -> {w,z} };
    third root -> child -> grandchild -> youngster -> third root;
  };
```

Here is another example, where a different layout method is used that is more appropriate for trees:




```
\usetikzlibrary {graphdrawing} \usegdlibrary {trees}
\tikz [grow'=up, binary tree layout, nodes={circle,draw}]
  \node {1}
  child { node {2}
    child { node {3} }
    child { node {4}
      child { node {5} }
      child { node {6} }
    }
  }
  child { node {7}
    child { node {8}
      child[missing]
      child { node {9} }
    }
  };
```

A final example, this time using a "spring electrical layout" (whatever that might be…):




```
\usetikzlibrary {decorations.pathmorphing,graphdrawing} \usegdlibrary {force}
\tikz [spring electrical layout, node distance=1.3cm,
       every edge/.style={
         decoration={coil, aspect=-.5, post length=1mm,
                     segment length=1mm, pre length=2mm},
         decorate, draw}]
{
  \foreach \i in {1,...,6}
    \node (node \i) [fill=blue!50, text=white, circle] {\i};

  \draw (node 1) edge (node 2)
        (node 2) edge (node 3)
                 edge (node 4)
        (node 3) edge (node 4)
                 edge (node 5)
                 edge (node 6);
}
```

In all of the example, the positions of the nodes have only been computed *after* all nodes have been created and the edges have been specified. For instance, in the last example, without the option `spring electrical layout`, all of the nodes would have been placed on top of each other.

27.3 Extending the Graph Drawing System

The graph drawing engine is also intended to make is (relatively) easy to implement new graph drawing algorithms. These algorithms can either be implemented in the Lua programming language (which is *much*

easier to program than TEX itself) or in C/C++ (but at a great cost regarding portability). The Lua code for a graph drawing algorithm gets an object-oriented model of the input graph as an input and must just compute the desired new positions of the nodes. The complete handling of passing options and configurations back-and-forth between the different Ti*k*Z and PGF layers is handled by the graph drawing engine.

As a caveat, the graph drawing engine comes with a library of functions and methods that simplify the writing of new graph drawing algorithms. As a typical example, when you implement a graph drawing algorithm for trees, you typically require that your input is a tree; but you can bet that users will feed all sorts of graphs to your algorithm, including disjoint unions of cliques. The graph drawing engine offers you to say that a precondition to running your algorithm is that the graph is a `tree` and instead of the original graph your algorithm will be provided with a spanning tree of the graph on which it can work. There are numerous further automatic pre- and postprocessing steps that include orienting, anchoring, and packing of components, to name a few.

The bottom line is that the graph drawing engine makes it easy to try out new graph drawing algorithms for medium sized graphs (up to a few hundred nodes) in Lua. For larger graphs, C/C++ code must be used.

27.4 The Layers of the Graph Drawing System

Even though the graph drawing system presented in the following sections was developed as part of PGF, it can be used independently of PGF and Ti*k*Z: It was (re)designed so that it can be used by arbitrary programs as long as they are able to run Lua. To achieve this, the graph drawing system consists of three layers:

1. At the "bottom" we have the *algorithmic layer*. This layer, written in Lua, contains all graph drawing algorithms. Interestingly, options must also be declared on this layer, so an algorithm together with all options it uses can and must be specified entirely on this layer. If you intend to implement a new graph drawing algorithm, you will only be interested in the functionality of this layer.

   Algorithm "communicate" with the graph drawing system through a well-defined interface, encapsulated in the class `InterfaceToAlgorithms`.

2. At the "top" we have the *display layer*. This layer is not actually part of the graph drawing system. Rather, it is a piece of software that "displays" graphs and Ti*k*Z is just one example of such a software. Another example might be a graph editor that uses the graph drawing system to lay out the graph it displays. Yet another example might be a command line tool for drawing graphs described in a file. Finally, you may also wish to use the graph drawing system as a simple subroutine for rendering graphs produced in a larger program.

   Since the different possible instantiations of the display layer are quite heterogeneous, all display layers must communicate with the graph drawing system through a special interface, encapsulated in the class `InterfaceToDisplay`.

   The main job of this class is to provide a set of methods for specifying that a graph has certain nodes and edges and that certain options have been set for them. However, this interface also allows you to query all options that have been declared by algorithms, including their documentation. This way, an editor or a command line tool can display a list of all graph drawing algorithms and how they can be configured.

3. The algorithm layer and the display layer are "bound together" through the *binding layer*. Most of the bookkeeping concerning the to-be-drawn graphs is done by the graph drawing system independently of which algorithm is used and also independently of which display layer is used, but some things are still specific to each display layer. For instance, some algorithms may create new nodes and the algorithms may then need to know how large these nodes will be. For this, the display layer must be "queried" during a run of the algorithm – and it is the job of the binding layer to achieve this callback.

   As a rule, the binding layer implements the "backward" communication from the graph drawing system back to the display layer, while the display layer's interface class provides only functions that are called from the display layer but which will not "talk back".

All of the files concerned with graph drawing reside in the `graphdrawing` subdirectory of `generic/pgf`.

27.5 Organisation of the Graph Drawing Documentation

The documentation of the graph drawing engine is structured as follows:

1. Following this overview section, the next section documents the graph drawing engine from "the Ti*k*Z user's point of view". No knowledge of Lua or algorithmic graph drawing is needed for this section, everyone who intends to use algorithmic graph drawing in Ti*k*Z may be interested in reading it.

2. You will normally only use Ti*k*Z's keys and commands in order to use the graph drawing system, but, internally, these keys call more basic PGF commands that do the "hard work" of binding the world of TEX boxes and macros to the object-oriented world of Lua. Section 29 explains how this works and which commands are available for authors of packages that directly need to use the graph drawing system inside PGF, avoiding the overhead incurred by Ti*k*Z.

   Most readers can safely skip this section.

3. The next sections detail which graph drawing algorithms are currently implemented as part of the Ti*k*Z distribution, see Sections 30 to 35.

4. Section 36 is addressed at readers who wish to implement their own graph drawing algorithms. For this, *no knowledge at all* of TEX programming is needed. The section explains the graph model used in Lua, the available libraries, the graph drawing pipeline, and everything else that is part of the Lua side of the engine.

5. Section 38 details the display layer of the graph drawing system. You should read this section if you wish to implement a new display system (that is, a non-TEX-based program) that intends to use the graph drawing system.

6. Section 39 explains how binding layers can be implemented. This section, too, is of interest only to readers who wish to write new display systems.

27.6 Acknowledgements

Graph drawing in Ti*k*Z began as a student's project under my supervision. Renée Ahrens, Olof-Joachim Frahm, Jens Kluttig, Matthias Schulz, and Stephan Schuster wrote the first prototype of a graph drawing system inside Ti*k*Z that uses LuaTEX for the implementation of graph drawing algorithms.

This first, early version was greatly extended on the algorithmic side by Jannis Pohlmann who wrote his Diploma thesis on graph drawing under my supervision. He implemented, in particular, the Sugiyama method (`layered layout`) and force based algorithms. Also, he rewrote some of the code of the prototype.

At some point it became apparent that the first implementation had a number of deficiencies, both concerning the structure, the interfaces, and (in particular) the performance. Because of this, I rewrote the code of the graph drawing system, both on the TEX side and on the Lua side in its current form. However, I would like to stress that without the work of the people mentioned above graph drawing in Ti*k*Z would not exist.

The documentation was written almost entirely by myself, though I did copy some paragraphs from Jannis's Diploma thesis, which I can highly recommend everyone to read.

In the future, I hope that other people will contribute algorithms, which will be available as libraries.

28 Using Graph Drawing in Ti*k*Z

by Till Tantau

Ti*k*Z Library graphdrawing

```
\usetikzlibrary{graphdrawing} % LaTeX and plain TeX
\usetikzlibrary[graphdrawing] % ConTeXt
```

This package provides capabilities for automatic graph drawing. It requires that the document is typeset using LuaTeX This package should work with LuaTeX 0.54 or higher.

28.1 Choosing a Layout and a Library

The graph drawing engine is initialized when you load the library `graphdrawing`. This library provides the basic framework for graph drawing, including all options and keys described in the present section. However, this library does *not* load any actual algorithms for drawing graphs. For this, you need to use the following command, which is defined by the `graphdrawing` library:

`\usegdlibrary{`⟨*list of libraries*⟩`}`

This command is used to load the special graph drawing libraries (the `gd` in the name of the command stands for "graph drawing"). The ⟨*list of libraries*⟩ is a comma-separated list of library written in the Lua programming language (which is why a special command is needed).

In detail, this command does the following. For each ⟨*name*⟩ in the ⟨*list of libraries*⟩ we do:

1. Check whether LuaTeX can call `require` on the library file `pgf.gd.`⟨*name*⟩`.library`. LuaTeX's usual file search mechanism will search the texmf-trees in the usual manner and the dots in the file name get converted into directory slashes.
2. If the above failed, try to `require` the string `pgf.gd.`⟨*name*⟩.
3. If this fails, try to `require` the string ⟨*name*⟩`.library`.
4. If this fails, try to `require` the string ⟨*name*⟩. If this fails, print an error message.

The net effect of the above is the following: Authors of graph drawing algorithms can bundle together multiple algorithms in a library by creating a `...xyz/library.lua` file that internally just calls `require` for all files containing declarations. On the other hand, if a graph drawing algorithm completely fits inside a single file, it can also be read directly using `\usegdlibrary`.

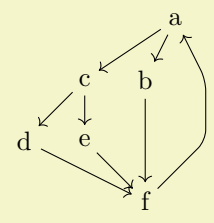
```
\usetikzlibrary{graphdrawing}
\usegdlibrary{trees,force}
```

The different graph drawing libraries are documented in the following Sections 30 to 35.

Note that in addition to the graph *drawing* libraries, you may also wish to load the normal Ti*k*Z library `graphs`. It provides the powerful `graph` path command with its easy-to-use syntax for specifying graphs, but you can use the graph drawing engine independently of the `graphs` library, for instance in conjunction with the `child` or the `edge` syntax. Here is a typical setup:

```
\usetikzlibrary{graphs, graphdrawing}
\usegdlibrary{trees, layered}
```

Having set things up, you must then specify for which scopes the graph drawing engine should apply a layout algorithm to the nodes in the scope. Typically, you just add an option ending with `... layout` to the `graph` path operation and then let the graph drawing do its magic:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz [rounded corners]
  \graph [layered layout, sibling distance=8mm, level distance=8mm]
  {
    a -> {
      b,
      c -> { d, e }
    } ->
    f ->
    a
  };
```

Whenever you use such an option, you can:

- Create nodes in the usual way. The nodes will be created completely, but then tucked away in an internal table. This means that all of Ti*k*Z's options for nodes can be applied. You can also name a node and reference it later.
- Create edges using either the syntax of the `graph` command (using `--`, `<-`, `->`, or `<->`), or using the `edge` command, or using the `child` command. These edges will, however, not be created immediately. Instead, the basic layer's command `\pgfgdedge` will be called, which stores "all the information concerning the edge". The actual drawing of the edge will only happen after all nodes have been positioned.
- Most of the keys that can be passed to an edge will work as expected. In particular, you can add labels to edges using the usual `node` syntax for edges.
- The `label` and `pin` options can be used in the usual manner with nodes inside a graph drawing scope. Only, the labels and nodes will play no role in the positioning of the nodes and they are added when the nodes are finally positioned.
- Similarly, nodes that are placed "on an edge" using the implicit positioning syntax can be used in the usual manner.

Here are some things that will *not* work:

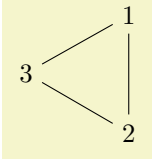
- Only edges created using the graph syntax, the `edge` command, or the `child` command will correctly pass their connection information to the basic layer. When you write `\draw (a)--(b);` inside a graph drawing scope, where `a` and `b` are nodes that have been created inside the scope, you will get an error message / things will look wrong. The reason is that the usual `--` is not "caught" by the graph drawing engine and, thus, tries to immediately connect two nodes that do not yet exist (except inside some internal table).
- The options of edges are executed twice: Once when the edge is "examined" by the `\pgfgdedge` command (using some magic to shield against the side effects) and then once more when the edge is actually created. Fortunately, in almost all cases, this will not be a problem; but if you do very evil magic inside your edge options, you must roll a D100 to see what strange things will happen. (Do no evil, by the way.)

If you are really interested in the "fine print" of what happens, please see Section 29.

28.2 Graph Drawing Parameters

Graph drawing algorithms can typically be configured in some way. For instance, for a graph drawing algorithm that visualizes its nodes as a tree, it will typically be useful when the user can change the so-called *level distance* and the *sibling distance*. For other algorithms, like force-based algorithms, a large number of parameters influence the way the algorithms work. Options that influence graph drawing algorithms will be called *(graph drawing) parameters* in the following. From the user's point of view, these parameters look like normal Ti*k*Z keys and you set them in the usual way. Internally, they are treated a bit differently from normal keys since their "effect" becomes apparent only later on, namely during the run of the graph drawing algorithm.

A graph drawing algorithm may or may not take different graph parameters into account. After all, these options may even outright contradict each other, so an algorithm can only try to "do its best". While many graph parameters are very specific to a single algorithm, a number of graph parameters will be important for many algorithms and they are documented in the course of the present section. Here is an example of an option the "always works":




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, vertical=1 to 2] { 1--2--3--1 };
```

28.3 Padding and Node Distances

In many drawings, you may wish to specify how "near" two nodes should be placed by a graph drawing algorithm. Naturally, this depends strongly on the specifics of the algorithm, but there are a number of general keys that will be used by many algorithms.

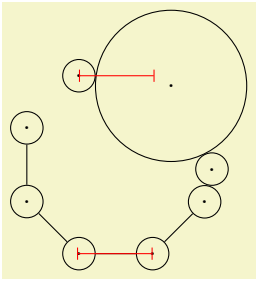
There are different kinds of objects for which you can specify distances and paddings:

- You specify the "natural" distance between nodes connected by an edge using `node distance`, which is also available in normal Ti*k*Z albeit for a slightly different purpose. However, not every algorithm will (or can) honor the key; see the description of each algorithm what it will "make of this option".
- A number of graph drawing algorithms arrange nodes in layers (or levels); we refer to the nodes on the same layer as siblings (although, in a tree, siblings are only nodes with the same parent; nevertheless we use "sibling" loosely also for nodes that are more like "cousins").
- When a graph consists of several connected component, many graph drawing algorithms will layout these components individually. The different components will then be arranged next to each other, see Section 28.7 for the details, such that between the nodes of any two components a padding is available.

`/graph drawing/node distance=`⟨*length*⟩ (initially `1cm`)

This is minimum distance that the centers of nodes connected by an edge should have. It will not always be possible to satisfy this desired distance, for instance in case the nodes are too big. In this case, the ⟨*length*⟩ is just considered as a lower bound.

Example

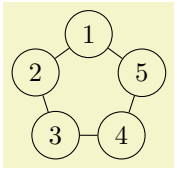


```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\begin{tikzpicture}
  \graph [simple necklace layout, node distance=1cm, node sep=0pt,
          nodes={draw,circle,as=.}]
  {
    1 -- 2 [minimum size=2cm] -- 3 --
    4 -- 5 -- 6 -- 7 --[orient=up] 8
  };
  \draw [red,|-|] (1.center) -- ++(0:1cm);
  \draw [red,|-|] (5.center) -- ++(180:1cm);
\end{tikzpicture}
```

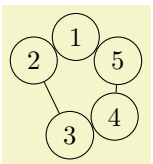
`/graph drawing/node pre sep=`⟨*length*⟩ (initially `.333em`)

This is a minimum "padding" or "separation" between the border of nodes connected by an edge. Thus, if nodes are so big that nodes with a distance of `node distance` would overlap (or just come with ⟨*dimension*⟩ distance of one another), their distance is enlarged so that this distance is still satisfied. The `pre` means that the padding is added to the node "at the front". This make sense only for some algorithms, like for a simple necklace layout.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout, node distance=0cm, nodes={circle,draw}]
  { 1--2--3--4--5--1 };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout, node distance=0cm, node sep=0mm,
              nodes={circle,draw}]
  { 1--2--3[node pre sep=5mm]--4--5[node pre sep=1mm]--1 };
```

/graph drawing/node post sep=⟨*length*⟩ (initially .333em)

Works like node pre sep.

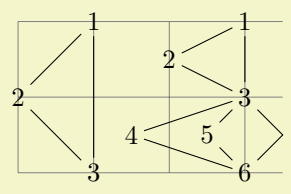
/graph drawing/node sep=⟨*length*⟩

A shorthand for setting both node pre sep and node post sep to ⟨*length*⟩/2.

/graph drawing/level distance=⟨*length*⟩ (initially 1cm)

This is minimum distance that the centers of nodes on one level should have from the centers of nodes on the next level. It will not always be possible to satisfy this desired distance, for instance in case the nodes are too big. In this case, the ⟨*length*⟩ is just considered as a lower bound.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}[inner sep=2pt]
  \draw [help lines] (0,0) grid (3.5,2);
  \graph [layered layout, level distance=1cm, level sep=0]
    { 1 [x=1,y=2] -- 2 -- 3 -- 1 };
  \graph [layered layout, level distance=5mm, level sep=0]
    { 1 [x=3,y=2] -- 2 -- 3 -- 1, 3 -- {4,5} -- 6 -- 3 };
\end{tikzpicture}
```

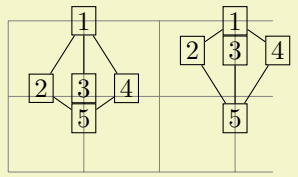
/graph drawing/layer distance=⟨*length*⟩

An alias for level distance

/graph drawing/level pre sep=⟨*length*⟩ (initially .333em)

This is a minimum "padding" or "separation" between the border of the nodes on a level to any nodes on the previous level. Thus, if nodes are so big that nodes on consecutive levels would overlap (or just come with ⟨*length*⟩ distance of one another), their distance is enlarged so that this distance is still satisfied. If a node on the previous level also has a level post sep, this post padding and the ⟨*dimension*⟩ add up. Thus, these keys behave like the "padding" keys rather than the "margin" key of cascading style sheets.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}[inner sep=2pt, level sep=0pt, sibling distance=0pt]
  \draw [help lines] (0,0) grid (3.5,2);
  \graph [layered layout, level distance=0cm, nodes=draw]
    { 1 [x=1,y=2] -- {2,3[level pre sep=1mm],4[level pre sep=5mm]} -- 5 };
  \graph [layered layout, level distance=0cm, nodes=draw]
    { 1 [x=3,y=2] -- {2,3,4} -- 5[level pre sep=5mm] };
\end{tikzpicture}
```

/graph drawing/level post sep=⟨*length*⟩ (initially .333em)

Works like level pre sep.

/graph drawing/layer pre sep=⟨*length*⟩

An alias for level pre sep.

/graph drawing/layer post sep=⟨*length*⟩

An alias for level post sep.

/graph drawing/level sep=⟨*length*⟩

A shorthand for setting both level pre sep and level post sep to ⟨*length*⟩/2. Note that if you set level distance=0 and level sep=1em, you get a layout where any two consecutive layers are "spaced apart" by 1em.

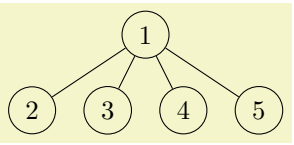
/graph drawing/layer sep=⟨*number*⟩

An alias for level sep.

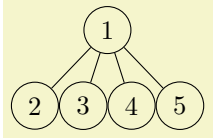
`/graph drawing/sibling distance=`⟨*length*⟩ (initially `1cm`)

This is minimum distance that the centers of node should have to the center of the next node on the same level. As for levels, this is just a lower bound. For some layouts, like a simple necklace layout, the ⟨*length*⟩ is measured as the distance on the circle.

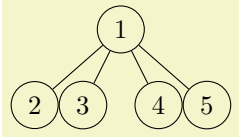
Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, sibling distance=1cm, nodes={circle,draw}]
  { 1--{2,3,4,5} };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, sibling distance=0cm, sibling sep=0pt,
              nodes={circle,draw}]
  { 1--{2,3,4,5} };
```

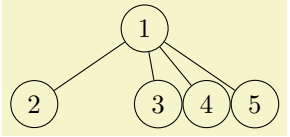



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, sibling distance=0cm, sibling sep=0pt,
              nodes={circle,draw}]
  { 1--{2,3[sibling distance=1cm],4,5} };
```

`/graph drawing/sibling pre sep=`⟨*length*⟩ (initially `.333em`)

Works like `level pre sep`, only for siblings.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, sibling distance=0cm, nodes={circle,draw},
              sibling sep=0pt]
  { 1--{2,3[sibling pre sep=1cm],4,5} };
```

`/graph drawing/sibling post sep=`⟨*length*⟩ (initially `.333em`)

Works like `sibling pre sep`.

`/graph drawing/sibling sep=`⟨*length*⟩

A shorthand for setting both `sibling pre sep` and `sibling post sep` to ⟨*length*⟩/2.

`/graph drawing/part distance=`⟨*length*⟩ (initially `1.5cm`)

This is minimum distance between the centers of "parts" of a graph. What a "part" is depends on the algorithm.

`/graph drawing/part pre sep=`⟨*length*⟩ (initially `1em`)

A pre-padding for parts.

`/graph drawing/part post sep=`⟨*length*⟩ (initially `1em`)

A post-padding for pars.

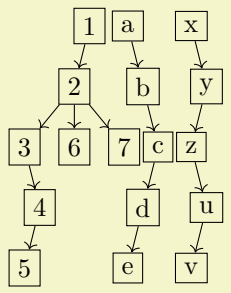
`/graph drawing/part sep=`⟨*length*⟩

A shorthand for setting both `part pre sep` and `part post sep` to ⟨*length*⟩/2.

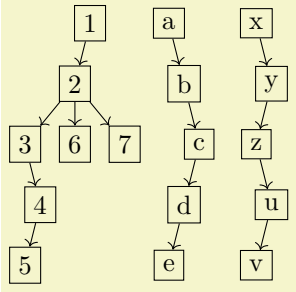
`/graph drawing/component sep=`⟨*length*⟩ (initially `1.5em`)

This is padding between the bounding boxes that nodes of different connected components will have when they are placed next to each other.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, sibling distance=4mm, level distance=8mm,
              components go right top aligned,
              component sep=1pt, nodes=draw]
{
  1 -> 2 -> {3->4[second]->5,6,7};
  a -> b[second] -> c[second] -> d -> e;
  x -> y[second] -> z -> u[second] -> v;
};
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, sibling distance=4mm, level distance=8mm,
              components go right top aligned,
              component sep=1em, nodes=draw]
{
  1 -> 2 -> {3->4[second]->5,6,7};
  a -> b[second] -> c[second] -> d -> e;
  x -> y[second] -> z -> u[second] -> v;
};
```

`/graph drawing/component distance=`⟨*length*⟩ (initially `2cm`)

This is the minimum distance between the centers of bounding boxes of connected components when they are placed next to each other. (Not used, currently.)

28.4 Anchoring a Graph

A graph drawing algorithm must compute positions of the nodes of a graph, but the computed positions are only *relative* ("this node is left of this node, but above that other node"). It is not immediately obvious where the "the whole graph" should be placed *absolutely* once all relative positions have been computed. In case that the graph consists of several unconnected components, the situation is even more complicated.

The order in which the algorithm layer determines the node at which the graph should be anchored:

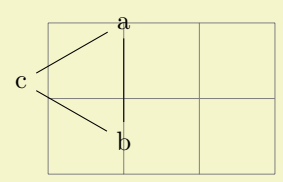
1. If the `anchor node=`⟨*node name*⟩ option given to the graph as a whole, the graph is anchored at ⟨*node name*⟩, provided there is a node of this name in the graph. (If there is no node of this name or if it is misspelled, the effect is the same as if this option had not been given at all.)
2. Otherwise, if there is a node where the `anchor here` option is specified, the first node with this option set is used.
3. Otherwise, if there is a node where the `desired at` option is set (perhaps implicitly through keys like `x`), the first such node is used.
4. Finally, in all other cases, the first node is used.

In the above description, the "first" node refers to the node first encountered in the specification of the graph.

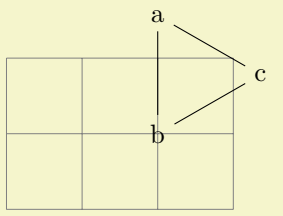
Once the node has been determined, the graph is shifted so that this node lies at the position specified by `anchor at`.

`/graph drawing/desired at=`⟨*coordinate*⟩

When you add this key to a node in a graph, you "desire" that the node should be placed at the ⟨*coordinate*⟩ by the graph drawing algorithm. Now, when you set this key for a single node of a graph, then, by shifting the graph around, this "wish" can obviously always be fulfill:

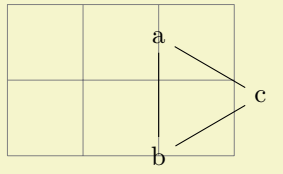



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a [desired at={(1,2)}] -- b -- c -- a;
  };
\end{tikzpicture}
```

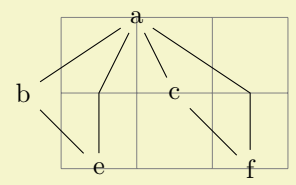



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a -- b[desired at={(2,1)}] -- c -- a;
  };
\end{tikzpicture}
```

Since the key's name is a bit long and since the many braces and parentheses are a bit cumbersome, there is a special support for this key inside a `graph`: The standard `/tikz/at` key is redefined inside a `graph` so that it points to `/graph drawing/desired at` instead. (Which is more logical anyway, since it makes no sense to specify an `at` position for a node whose position it to be computed by a graph drawing algorithm.) A nice side effect of this is that you can use the `x` and `y` keys (see Section 19.9.1) to specify desired positions:



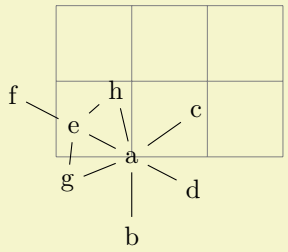

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a -- b[x=2,y=0] -- c -- a;
  };
\end{tikzpicture}
```



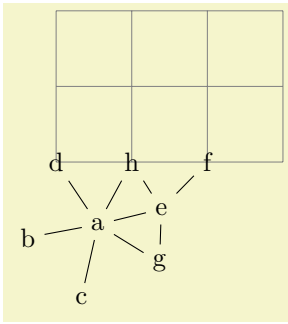

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [layered layout]
  {
    a [x=1,y=2] -- { b, c } -- {e, f} -- a
  };
\end{tikzpicture}
```

A problem arises when two or more nodes have this key set, because then your "desires" for placement and the positions computed by the graph drawing algorithm may clash. Graph drawing algorithms are "told" about the desired positions. Most algorithms will simply ignore these desired positions since they will be taken care of in the so-called post-anchoring phase, see below. However, for some algorithms it makes a lot of sense to fix the positions of some nodes and only compute the positions of the other nodes relative to these nodes. For instance, for a `spring layout` it makes perfect sense that some nodes are "nailed to the canvas" while other nodes can "move freely".

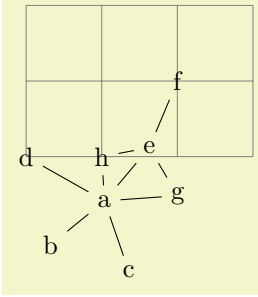
Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a[x=1] -- { b, c, d, e -- {f,g,h} };
    { h, g } -- a;
  };
\end{tikzpicture}
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a -- { b, c, d[x=0], e -- {f[x=2], g, h[x=1]} };
    { h, g } -- a;
  };
\end{tikzpicture}
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \graph [spring layout]
  {
    a -- { b, c, d[x=0], e -- {f[x=2,y=1], g, h[x=1]} };
    { h, g } -- a;
  };
\end{tikzpicture}
```

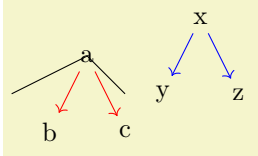
`/graph drawing/anchor node=`⟨*string*⟩

This option can be used with a graph to specify a node that should be used for anchoring the whole graph. When this option is specified, after the layout has been computed, the whole graph will be shifted in such a way that the ⟨*node name*⟩ is either

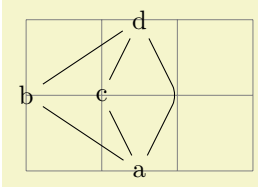
- at the current value of `anchor at` or
- at the position that is specified in the form of a `desired at` for the ⟨*node name*⟩.

Note how in the last example `c` is placed at `(1,1)` rather than `b` as would happen by default.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \draw (0,0)
  -- (1,0.5) graph [edges=red,  layered layout, anchor node=a] { a -> {b,c} }
  -- (1.5,0) graph [edges=blue, layered layout,
                    anchor node=y, anchor at={(2,0)}]          { x -> {y,z} };
```



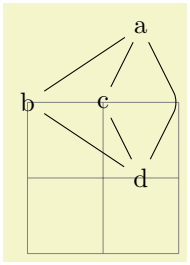

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \graph [layered layout, anchor node=c, edges=rounded corners]
    { a -- {b [x=1,y=1], c [x=1,y=1] } -- d -- a};
\end{tikzpicture}
```

`/graph drawing/anchor at=`⟨*canvas coordinate*⟩ (initially `(0pt,0pt)`)

The coordinate at which the graph should be anchored when no explicit anchor is given for any node. The initial value is the origin.

Example



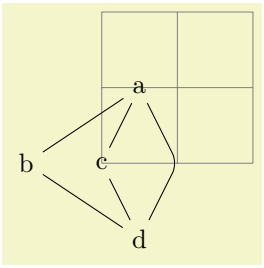

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);

  \graph [layered layout, edges=rounded corners, anchor at={(1,2)}]
    { a -- {b, c [anchor here] } -- d -- a};
\end{tikzpicture}
```

`/graph drawing/anchor here=`⟨*boolean*⟩ (default `true`)

This option can be passed to a single node (rather than the graph as a whole) in order to specify that this node should be used for the anchoring process. In the example, `c` is placed at the origin since this is the default `anchor at` position.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);

  \graph [layered layout, edges=rounded corners]
    { a -- {b, c [anchor here] } -- d -- a};
\end{tikzpicture}
```

28.5 Orienting a Graph

Just as a graph drawing algorithm cannot know *where* a graph should be placed on a page, it is also often unclear which *orientation* it should have. Some graphs, like trees, have a natural direction in which they "grow", but for an "arbitrary" graph the "natural orientation" is, well, arbitrary.

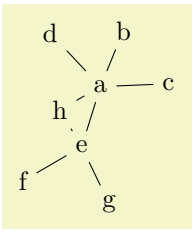
There are two ways in which you can specify an orientation: First, you can specify that the line from a certain vertex to another vertex should have a certain slope. How these vertices and slopes are specified in explained momentarily. Second, you can specify a so-called "growth direction" for trees.

`/graph drawing/orient=`⟨*direction*⟩ (default `0`)

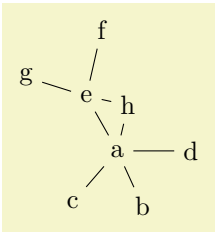
This key specifies that the straight line from the `orient tail` to the `orient head` should be at an angle of ⟨*direction*⟩ relative to the right-going x-axis. Which vertices are used as tail an head depends on where the `orient` option is encountered: When used with an edge, the tail is the edge's tail and the head is the edge's head. When used with a node, the tail or the head must be specified explicitly and the node is used as the node missing in the specification. When used with a graph as a whole, both the head and tail nodes must be specified explicitly. Note that the ⟨*direction*⟩ is independent of the actual to-path of an edge, which might define a bend or more complicated shapes. For instance, a ⟨*angle*⟩ of `45` requests that the end node is "up and right" relative to the start node.

You can also specify the standard direction texts `north` or `south east` and so forth as ⟨*direction*⟩ and also `up`, `down`, `left`, and `right`. Also, you can specify `-` for "right" and `|` for "down".

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]
{
  a -- { b, c, d, e -- {f, g, h} };
  h -- [orient=30] a;
};
```

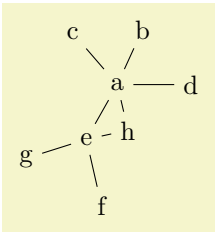



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]
{
  a -- { b, c, d[> orient=right], e -- {f, g, h} };
  h -- a;
};
```

`/graph drawing/orient'=`⟨*direction*⟩ (default 0)

Same as `orient`, only the rest of the graph should be flipped relative to the connection line.

Example

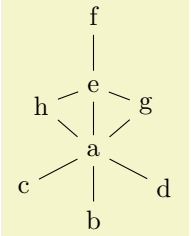



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]
{
  a -- { b, c, d[> orient'=right], e -- {f, g, h} };
  h -- a;
};
```

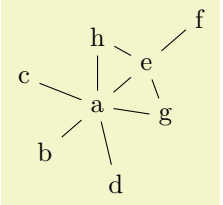
`/graph drawing/orient tail=`⟨*string*⟩

Specifies the tail vertex for the orientation of a graph. See `orient` for details.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout] {
  a [orient=/, orient tail=f] -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```

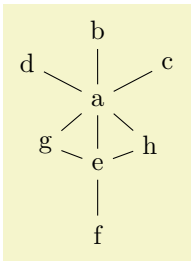



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout] {
  a [orient=down, orient tail=h] -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```

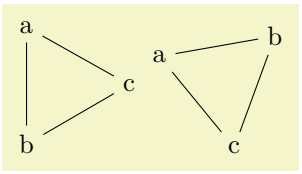
`/graph drawing/orient head=`⟨*string*⟩

Specifies the head vertex for the orientation of a graph. See `orient` for details.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]
{
  a [orient=/, orient head=f] -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```

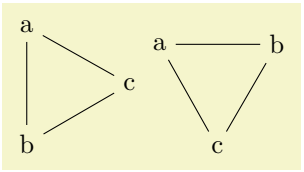



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout] { a -- b -- c -- a };
\tikz \graph [spring layout, orient=10,
              orient tail=a, orient head=b] { a -- b -- c -- a };
```

`/graph drawing/horizontal=`⟨*string*⟩

A shorthand for specifying `orient tail`, `orient head` and `orient=0`. The tail will be everything before the part “ `to` ” and the head will be everything following it.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]                    { a -- b -- c -- a };
\tikz \graph [spring layout, horizontal=a to b] { a -- b -- c -- a };
```

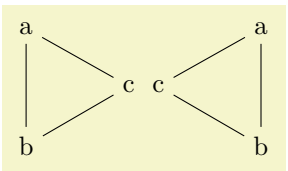
`/graph drawing/horizontal'=`⟨*string*⟩

Like `horizontal`, but with a flip.

`/graph drawing/vertical=`⟨*string*⟩

A shorthand for specifying `orient tail`, `orient head` and `orient=-90`.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout]                    { a -- b -- c -- a };
\tikz \graph [spring layout, vertical=a to b] { a -- b -- c -- a };
```

`/graph drawing/vertical'=`⟨*string*⟩

Like `vertical`, but with a flip.

`/graph drawing/grow=`⟨*direction*⟩

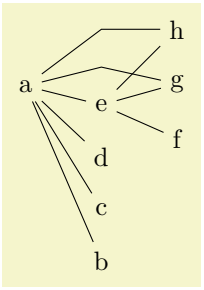
This key specifies in which direction the neighbors of a node “should grow”. For some graph drawing algorithms, especially for those that layout trees, but also for those that produce layered layouts, there is a natural direction in which the “children” of a node should be placed. For instance, saying `grow=down` will cause the children of a node in a tree to be placed in a left-to-right line below the node (as always,

you can replace the ⟨*angle*⟩ by direction texts). The children are requested to be placed in a counter-clockwise fashion, the `grow'` key will place them in a clockwise fashion. Note that when you say `grow=down`, it is not necessarily the case that any particular node is actually directly below the current node; the key just requests that the direction of growth is downward.

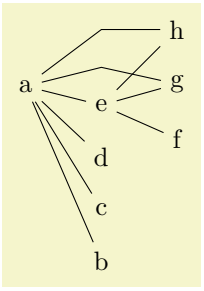
In principle, you can specify the direction of growth for each node individually, but do not count on graph drawing algorithms to honor these wishes.

When you give the `grow=right` key to the graph as a whole, it will be applied to all nodes. This happens to be exactly what you want:

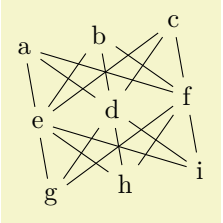
Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout, sibling distance=5mm]
{
  a [grow=right] -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout, grow=right, sibling distance=5mm]
{
  a -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```

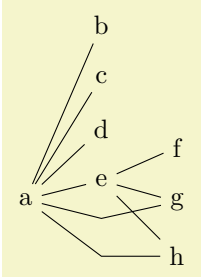



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz
  \graph [layered layout, grow=-80]
  {
    {a,b,c} --[complete bipartite] {e,d,f}
            --[complete bipartite] {g,h,i};
  };
```

`/graph drawing/grow'=`⟨*direction*⟩

Same as `grow`, only with the children in clockwise order.

Example



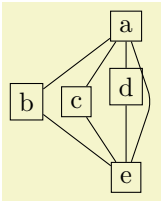

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout, sibling distance=5mm]
{
  a [grow'=right] -- { b, c, d, e -- {f, g, h} };
  { h, g } -- a;
};
```

28.6 Fine-Tuning Positions of Nodes

`/graph drawing/nudge=`⟨*canvas coordinate*⟩

This option allows you to slightly "nudge" (move) nodes after they have been positioned by the given offset. The idea is that this nudging is done after the position of the node has been computed, so nudging has no influence on the actual graph drawing algorithms. This, in turn, means that you can use nudging to "correct" or "optimize" the positioning of nodes after the algorithm has computed something.

Example

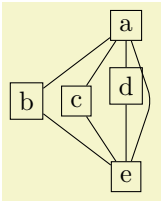



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [edges=rounded corners, nodes=draw,
              layered layout, sibling distance=0] {
  a -- {b, c, d[nudge=(up:2mm)]} -- e -- a;
};
```

`/graph drawing/nudge up=`⟨*length*⟩

A shorthand for nudging a node upwards.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [edges=rounded corners, nodes=draw,
              layered layout, sibling distance=0] {
  a -- {b, c, d[nudge up=2mm]} -- e -- a;
};
```

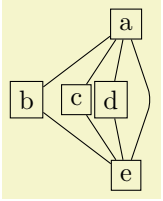
`/graph drawing/nudge down=`⟨*length*⟩

Like `nudge up`, but downwards.

`/graph drawing/nudge left=`⟨*length*⟩

Like `nudge up`, but left.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [edges=rounded corners, nodes=draw,
              layered layout, sibling distance=0] {
  a -- {b, c, d[nudge left=2mm]} -- e -- a;
};
```

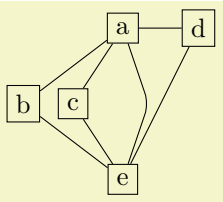
`/graph drawing/nudge right=`⟨*length*⟩

Like `nudge left`, but right.

`/graph drawing/regardless at=`⟨*canvas coordinate*⟩

Using this option you can provide a position for a node to wish it will be forced after the graph algorithms have run. So, the node is positioned normally and the graph drawing algorithm does not know about the position specified using `regardless at`. However, afterwards, the node is placed there, regardless of what the algorithm has computed (all other nodes are unaffected).

Example

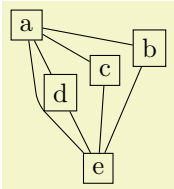



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [edges=rounded corners, nodes=draw,
              layered layout, sibling distance=0] {
  a -- {b,c,d[regardless at={(1,0)}]} -- e -- a;
};
```

`/graph drawing/nail at=`⟨*canvas coordinate*⟩

This option combines `desired at` and `regardless at`. Thus, the algorithm is "told" about the desired position. If it fails to place the node at the desired position, it will be put there regardless. The name of the key is intended to remind one of a node being "nailed" to the canvas.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [edges=rounded corners, nodes=draw,
              layered layout, sibling distance=0] {
  a -- {b,c,d[nail at={(1,0)}]} -- e[nail at={(1.5,-1)}] -- a;
};
```

28.7 Packing of Connected Components

Graphs may be composed of subgraphs or *components* that are not connected to each other. In order to draw these nicely, most graph drawing algorithms split them into separate graphs, compute their layouts with the same graph drawing algorithm independently and, in a postprocessing step, arrange them next to each other. Note, however, that some graph drawing algorithms can also arrange the nodes of the graph in a uniform way even for unconnected components (the `simple necklace layout` is a case in point); for such algorithms you can choose whether they should be applied to each component individually or not (if not, the following options do not apply). To configure which is the case, use the `componentwise` key.

The default method for placing the different components works as follows:

1. For each component, a layout is determined and the component is oriented as described Section 28.5 on the orientation of graphs.
2. The components are sorted as prescribed by the `component order` key.
3. The first component is now placed (conceptually) at the origin. (The final position of this and all other components will be determined later, namely in the anchoring phase, but let us imagine that the first component lies at the origin at this point.)
4. The second component is now positioned relative to the first component. The "direction" in which the next component is placed relative to the first one is determined by the `component direction` key, so components can be placed from left to right or up to down or in any other direction (even something like 30°). However, both internally and in the following description, we assume that the components are placed from left to right; other directions are achieved by doing some (clever) rotating of the arrangement achieved in this way.

   So, we now wish to place the second component to the right of the first component. The component is first shifted vertically according to some alignment strategy. For instance, it can be shifted so that the topmost node of the first component and the topmost node of the second component have the same vertical position. Alternatively, we might require that certain "alignment nodes" in both components have the same vertical position. There are several other strategies, which can be configured using the `component align` key.

   One the vertical position has been fixed, the horizontal position is computed. Here, two different strategies are available: First, image rectangular bounding boxed to be drawn around both components. Then we shift the second component such that the right border of the bounding box of the first component touches the left border of the bounding box of the second component. Instead of having

the bounding boxes "touch", we can also have a padding of `component sep` between them. The second strategy is more involved and also known as a "skyline" strategy, where (roughly) the components are "moved together as near as possible so that nodes do not touch".

5. After the second component has been placed, the third component is considered and positioned relative to the second one, and so on.

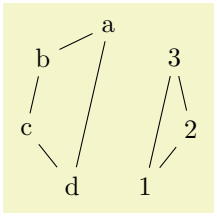
6. At the end, as hinted at earlier, the whole arrangement is rotate so that instead of "going right" the component go in the direction of `component direction`. Note, however, that this rotation applies only to the "shift" of the components; the components themselves are not rotated. Fortunately, this whole rotation process happens in the background and the result is normally exactly what you would expect.

In the following, we go over the different keys that can be used to configure the component packing.

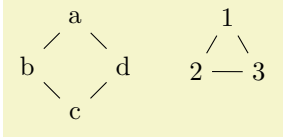
`/graph drawing/componentwise=`⟨*boolean*⟩ (default `true`)

For algorithms that also support drawing unconnected graphs, use this key to enforce that the components of the graph are, nevertheless, laid out individually. For algorithms that do not support laying out unconnected graphs, this option has no effect; rather it works as if this option were always set.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout]
  {
    a -- b -- c -- d -- a,
    1 -- 2 -- 3 -- 1
  };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout, componentwise]
  {
    a -- b -- c -- d -- a,
    1 -- 2 -- 3 -- 1
  };
```

28.7.1 Ordering the Components

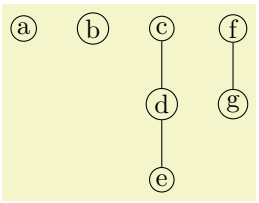
The different connected components of the graph are collected in a list. The ordering of the nodes in this list can be configured using the following key.

`/graph drawing/component order=`⟨*string*⟩ (initially `by first specified node`)

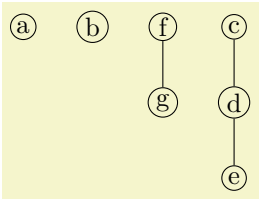
Selects a "strategy" for ordering the components. By default, they are ordered in the way they appear in the input. The following values are permissible for ⟨*strategy*⟩

- `by first specified node`
  The components are ordered "in the way they appear in the input specification of the graph". More precisely, for each component consider the node that is first encountered in the description of the graph. Order the components in the same way as these nodes appear in the graph description.
- `increasing node number`
  The components are ordered by increasing number of nodes. For components with the same number of nodes, the first node in each component is considered and they are ordered according to the sequence in which these nodes appear in the input.
- `decreasing node number` As above, but in decreasing order.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              component order=by first specified node]
  { a, b, c, f -- g, c -- d -- e };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              component order=increasing node number]
  { a, b, c -- d -- e, f -- g };
```

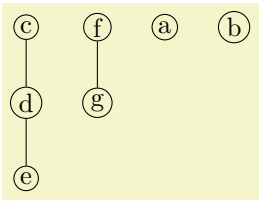
`/graph drawing/small components first=`⟨*string*⟩

A shorthand for `component order=increasing node number`.

`/graph drawing/large components first=`⟨*string*⟩

A shorthand for `component order=decreasing node number`.

Example



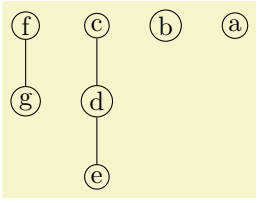

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              large components first]
  { a, b, c -- d -- e, f -- g };
```

28.7.2 Arranging Components in a Certain Direction

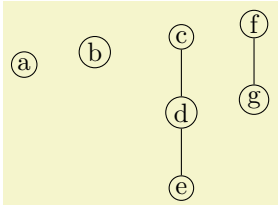
`/graph drawing/component direction=`⟨*direction*⟩ (initially `0`)

The ⟨*angle*⟩ is used to determine the relative position of each component relative to the previous one. The direction need not be a multiple of `90`. As usual, you can use texts like `up` or `right` instead of a number. As the examples show, the direction only has an influence on the relative positions of the components, not on the direction of growth inside the components. In particular, the components are not rotated by this option in any way. You can use the `grow` option or `orient` options to orient individual components.

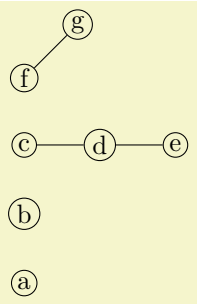
Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              component direction=left]
  { a, b, c -- d -- e, f -- g };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              component direction=10]
  { a, b, c -- d -- e, f -- g };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={inner sep=1pt,draw,circle},
              component direction=up]
   { a, b, c [grow=right] -- d -- e, f[grow=45] -- g };
```

28.7.3 Aligning Components

When components are placed next to each from left to right, it is not immediately clear how the components should be aligned vertically. What happens is that in each component a horizontal line is determined and then all components are shifted vertically so that the lines are aligned. There are different strategies for choosing these "lines", see the description of the options described later on. When the `component direction` option is used to change the direction in which components are placed, it certainly make no longer sense to talk about "horizontal" and "vertical" lines. Instead, what actually happens is that the alignment does not consider "horizontal" lines, but lines that go in the direction specified by `component direction` and aligns them by moving components along a line that is perpendicular to the line. For these reasons, let us call the line in the component direction the *alignment line* and a line that is perpendicular to it the *shift line*.

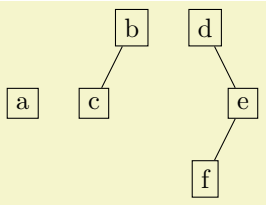
The first way of specifying through which point of a component the alignment line should get is to use the option `align here`. In many cases, however, you will not wish to specify an alignment node manually in each component. Instead, you will use the `component align` key to specify a *strategy* that should be used to automatically determine such a node.

Using a combination of `component direction` and `component align`, numerous different packing strategies can be configured. However, since names like `counterclockwise` are a bit hard to remember and to apply in practice, a number of easier-to-remember keys are predefined that combine an alignment and a direction.

`/graph drawing/align here=`⟨*boolean*⟩ (default `true`)

When this option is given to a node, this alignment line will go through the origin of this node. If this option is passed to more than one node of a component, the node encountered first in the component is used.

Example

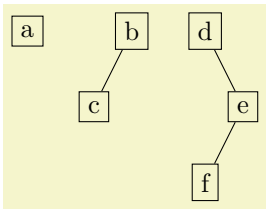



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw}]
  { a, b -- c[align here], d -- e[second, align here] -- f };
```

`/graph drawing/component align=`⟨*string*⟩ (initially `first node`)

Specifies a "strategy" for the alignment of components. The following values are permissible:

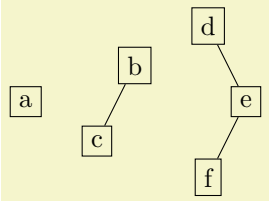
- `first node`
  In each component, the alignment line goes through the center of the first node of the component encountered during specification of the component.



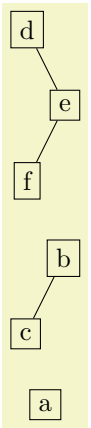

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component align=first node]
  { a, b -- c, d -- e[second] -- f };
```

- `center`

  The nodes of the component are projected onto the shift line. The alignment line is now chosen so that it is exactly in the middle between the maximum and minimum value that the projected nodes have on the shift line.



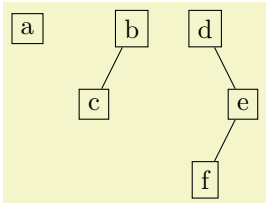

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component align=center]
  { a, b -- c, d -- e[second] -- f };
```



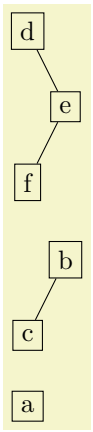

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component direction=90,
              component align=center]
  { a, b -- c, d -- e[second] -- f };
```

- `counterclockwise`

  As for `center`, we project the nodes of the component onto the shift line. The alignment line is now chosen so that it goes through the center of the node whose center has the highest projected value.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component align=counterclockwise]
  { a, b -- c, d -- e[second] -- f };
```

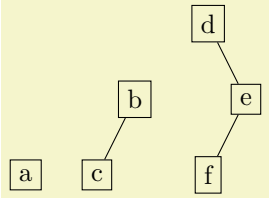



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component direction=90,
              component align=counterclockwise]
 { a, b -- c, d -- e[second] -- f };
```

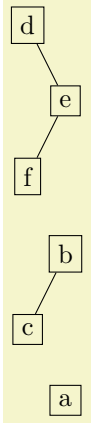
The name `counterclockwise` is intended to indicate that the align line goes through the node that comes last if we go from the alignment direction in a counter-clockwise direction.

- `clockwise`

  Works like `counterclockwise`, only in the other direction:



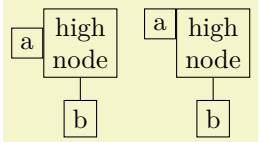

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component align=clockwise]
  { a, b -- c, d -- e[second] -- f };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, nodes={draw},
              component direction=90,
              component align=clockwise]
  { a, b -- c, d -- e[second] -- f };
```

- `counterclockwise bounding box`

  This method is quite similar to `counterclockwise`, only the alignment line does not go through the center of the node with a maximum projected value on the shift line, but through the maximum value of the projected bounding boxes. For a left-to-right packing, this means that the components are aligned so that the bounding boxes of the components are aligned at the top.

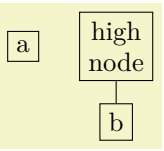



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
              component sep=0pt,
              component align=counterclockwise]
  { a, "high\\node" -- b};\quad
\tikz \graph [tree layout, nodes={draw, align=center},
              component sep=0pt,
              component align=counterclockwise bounding box]
  { a, "high\\node" -- b};
```

- `clockwise bounding box`
  Works like `counterclockwise bounding box`.

`/graph drawing/components go right top aligned=`⟨*string*⟩
Shorthand for `component direction=right` and `component align=counterclockwise`. This means that, as the name suggest, the components will be placed left-to-right and they are aligned such that their top nodes are in a line.

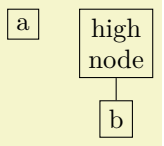
Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
                components go right top aligned]
  { a, "high\\node" -- b};
```

`/graph drawing/components go right absolute top aligned=`⟨*string*⟩
Like `components go right top aligned`, but with `component align` set to `counterclockwise bounding box`. This means that the components will be aligned with their bounding boxed being top-aligned.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
                components go right absolute top aligned]
  { a, "high\\node" -- b};
```

`/graph drawing/components go right bottom aligned=`⟨*string*⟩
See the other `components go ...` keys.

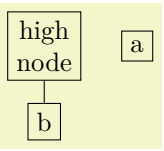
`/graph drawing/components go right absolute bottom aligned=`⟨*string*⟩
See the other `components go ...` keys.

`/graph drawing/components go right center aligned=`⟨*string*⟩
See the other `components go ...` keys.

`/graph drawing/components go right=`⟨*string*⟩
Shorthand for `component direction=right` and `component align=first node`.

`/graph drawing/components go left top aligned=`⟨*string*⟩
See the other `components go ...` keys.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
                components go left top aligned]
  { a, "high\\node" -- b};
```

`/graph drawing/components go left absolute top aligned=`⟨*string*⟩
See the other `components go ...` keys.

`/graph drawing/components go left bottom aligned=`⟨*string*⟩
See the other `components go ...` keys.

/graph drawing/components go left absolute bottom aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go left center aligned=⟨*string*⟩

See the other components go ... keys.

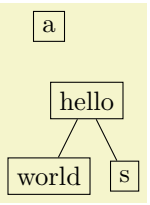
/graph drawing/components go left=⟨*string*⟩

See the other components go ... keys.

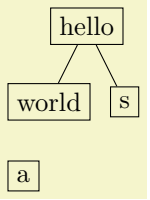
/graph drawing/components go down right aligned=⟨*string*⟩

See the other components go ... keys.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
                components go down left aligned]
  { a, hello -- {world,s} };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw, align=center},
                components go up absolute left aligned]
  { a, hello -- {world,s}};
```

/graph drawing/components go down absolute right aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go down left aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go down absolute left aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go down center aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go down=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up right aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up absolute right aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up left aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up absolute left aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up center aligned=⟨*string*⟩

See the other components go ... keys.

/graph drawing/components go up=⟨*string*⟩

See the other components go ... keys.

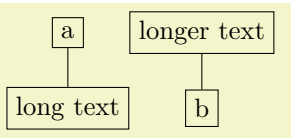
28.7.4 The Distance Between Components

Once the components of a graph have been oriented, sorted, aligned, and a direction has been chosen, it remains to determine the distance between adjacent components. Two methods are available for computing this distance, as specified by the following option:

`/graph drawing/component packing=`⟨*string*⟩ (initially `skyline`)

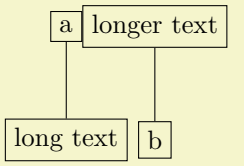
Given two components, their distance is computed as follows in dependence of ⟨*method*⟩:

- `rectangular`
  Imagine a bounding box to be drawn around both components. They are then shifted such that the padding (separating distance) between the two boxes is the current value of `component sep`.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
    \tikz \graph [tree layout, nodes={draw}, component sep=0pt,
                  component packing=rectangular]
      { a -- long text, longer text -- b};
```

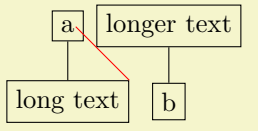
- `skyline`
  The "skyline method" is used to compute the distance. It works as follows: For simplicity, assume that the component direction is right (other case work similarly, only everything is rotated). Imaging the second component to be placed far right beyond the first component. Now start moving the second component back to the left until one of the nodes of the second component touches a node of the first component, and stop. Again, the padding `component sep` can be used to avoid the nodes actually touching each other.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
    \tikz \graph [tree layout, nodes={draw}, component sep=0pt,
                  level distance=1.5cm,
                  component packing=skyline]
      { a -- long text, longer text -- b};
```

  In order to avoid nodes of the second component "passing through a hole in the first component", the actual algorithm is a bit more complicated: For both components, a "skyline" is computed. For the first component, consider an arbitrary horizontal line. If there are one or more nodes on this line, the rightmost point on any of the bounding boxes of these nodes will be the point on the skyline of the first component for this line. Similarly, for the second component, for each horizontal level the skyline is given by the leftmost point on any of the bounding boxes intersecting the line.

  Now, the interesting case are horizontal lines that do not intersect any of the nodes of the first and/or second component. Such lines represent "holes" in the skyline. For them, the following rule is used: Move the horizontal line upward and downward as little as possible until a height is reached where there is a skyline defined. Then the skyline position on the original horizontal line is the skyline position at the reached line, minus (or, for the second component, plus) the distance by which the line was moved. This means that the holes are "filled up by slanted roofs".




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
    \begin{tikzpicture}
      \graph [tree layout, nodes={draw}, component sep=0pt,
              component packing=skyline]
      { a -- long text, longer text -- b};
      \draw[red] (long text.north east) -- ++(north west:1cm);
    \end{tikzpicture}
```

28.8 Anchoring Edges

When a graph has been laid out completely, the edges between the nodes must be drawn. Conceptually, an edge is "between two nodes", but when we actually draw the node, we do not really want the edge's path to start "in the middle" of the node; rather, we want it to start "on the border" and also end there.

Normally, computing such border positions for nodes is something we would leave to the so-called display layer (which is typically Ti*k*Z and Ti*k*Z is reasonably good at computing border positions). However, different display layers may behave differently here and even Ti*k*Z fails when the node shapes are very involved and the paths also.

For these reasons, computing the anchor positions where edges start and end is done inside the graph drawing system. As a user, you specify a `tail anchor` and a `head anchor`, which are points inside the tail and head nodes, respectively. The edge path will then start and end at these points, however, they will usually be shortened so that they actually start and end on the intersection of the edge's path with the nodes' paths.

`/graph drawing/tail anchor=`⟨*string*⟩

Specifies where in the tail vertex the edge should start. This is either a string or a number, interpreted as an angle (with 90 meaning "up"). If it is a string, when the start of the edge is computed, we try to look up the anchor in the tail vertex's table of anchors (some anchors get installed in this table by the display system). If it is not found, we test whether it is one of the special "direction anchors" like `north` or `south east`. If so, we convert them into points on the border of the node that lie in the direction of a line starting at the center to a point on the bounding box of the node in the designated direction. Finally, if the anchor is a number, we use a point on the border of the node that is on a line from the center in the specified direction.

If the anchor is set to the empty string (which is the default), the anchor is interpreted as the `center` anchor inside the graph drawing system. However, a display system may choose to make a difference between the `center` anchor and an empty anchor (Ti*k*Z does: for options like `bend left` if the anchor is empty, the bend line starts at the border of the node, while for the anchor set explicitly to `center` it starts at the center).

Note that graph drawing algorithms need not take the setting of this option into consideration. However, the final rendering of the edge will always take it into consideration (only, the setting may not be very sensible if the algorithm ignored it).

`/graph drawing/head anchor=`⟨*string*⟩

See `tail anchor`

`/graph drawing/tail cut=`⟨*boolean*⟩ (default `true`, initially `true`)

Decides whether the tail of an edge is "cut", meaning that the edge's path will be shortened at the beginning so that it starts only of the node's border rather than at the exact position of the `tail anchor`, which may be inside the node.

`/graph drawing/head cut=`⟨*boolean*⟩ (default `true`, initially `true`)

See `tail cut`.

`/graph drawing/cut policy=`⟨*string*⟩ (initially `as edge requests`)

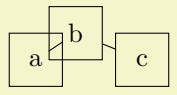
The policy for cutting edges entering or leaving a node. This option is important for nodes only. It can have three possible values:

- `as edge requests` Whether or not an edge entering or leaving the node is cut depends on the setting of the edge's `tail cut` and `head cut` options. This is the default.
- `all` All edges entering or leaving the node are cut, regardless of the edges' cut values.
- `none` No edge entering or leaving the node is cut, regardless of the edges' cut values.

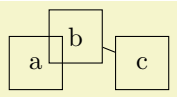
`/graph drawing/allow inside edges=`⟨*boolean*⟩ (default `true`, initially `true`)

Decides whether an edge between overlapping nodes should be drawn. If two vertices overlap, it may happen that when you draw an edge between them, this edges would be completely inside the two vertices. In this case, one could either not draw them or one could draw a sort of "inside edge".

Examples




```
\usetikzlibrary {graphs,graphdrawing}
\tikz \graph [no layout, nodes={draw, minimum size=20pt}] {
  a [x=0, y=0] -- b [x=15pt, y=10pt] -- c[x=40pt]
};
```




```
\usetikzlibrary {graphs,graphdrawing}
\tikz \graph [no layout, nodes={draw, minimum size=20pt},
              allow inside edges=false] {
  a [x=0, y=0] -- b [x=15pt, y=10pt] -- c[x=40pt]
};
```

28.9 Hyperedges

`/graph drawing/hyper` (style)

A *hyperedge* of a graph does not connect just two nodes, but is any subset of the node set (although a normal edge is also a hyperedge that happens to contain just two nodes). Internally, a collection of kind `hyper` is created. Currently, there is no default renderer for hyper edges.

```
\graph {
  % The nodes:
  a, b, c, d;

  % The edges:
  {[hyper] a,b,c};
  {[hyper] b,c,d};
  {[hyper] a,c};
  {[hyper] d}
};
```

28.10 Using Several Different Layouts to Draw a Single Graph

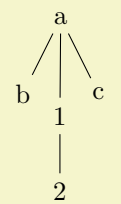
Inside each graph drawing scope, a main algorithm is used to perform the graph drawing. However, parts of the graph may be drawn using different algorithms: For instance, a graph might consist of several, say, cliques that are arranged in a tree-like fashion. In this case, it might be useful to layout each clique using a circular layout, but then lay out all laid out cliques using a tree drawing algorithm.

In order to lay out a graph using multiple algorithms, we need two things: First, we must be able to *specify* which algorithms should be used where and, second, we must be able to *resolve* conflicts that may result from different algorithms "having different ideas" concerning where nodes should be placed.

28.10.1 Sublayouts

Specifying different layouts for a graph is easy: Inside a graph drawing scope, simply open scopes, in which you use an option like `tree layout` for the nodes mentioned in this scope. Inside these scopes, you can open even subscopes for sublayouts, and so on. Furthermore, the `graphs` library has special support for sublayouts.

Let us start with the "plain" syntax for opening sublayouts: You pass a key for creating layouts to a `scope`:




```
\usetikzlibrary {graphdrawing} \usegdlibrary {force,trees}
\tikz [spring layout] {
  \begin{scope}[tree layout]
    \node (a) {a};
    \node (b) {b};
    \node (c) {c};
    \draw (a) edge (b) edge (c);
  \end{scope}

  \begin{scope}[tree layout]
    \node (1) {1};
    \node (2) {2};
    \draw (1) edge (2);
  \end{scope}

  \draw (a) edge (1);
}
```

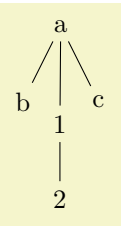
Let us see, what is going on here. The main layout (`spring layout`) contains two sublayouts (the two `tree layouts`). Both of them are laid out independently (more on the details in a moment). Then, from the main layout's point of view, the sublayouts behave like "large nodes" and, thus, the edge between `a` and

`1` is actually the only edge that is used by the `spring layout` – resulting in a simple layout consisting of one big node at the top and a big node at the bottom.

The `graphs` library has a special support for sublayouts: The syntax is as follows: wherever a normal node would go, you can write

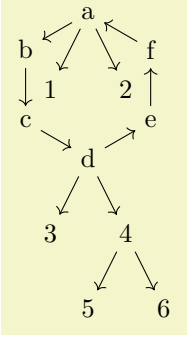
`// [`⟨*layout options*⟩`] {`⟨*sublayout*⟩`}`

Following the double slash, you may provide ⟨*layout options*⟩ in square brackets. However, you *must* provide a sublayout in braces. The contents of ⟨*sublayout*⟩ will be parsed using the usual `graph` syntax, but will form a sublayout.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force,trees}
\tikz \graph [spring layout] {
  // [tree layout] { a -- {b, c} };
  // [tree layout] { 1 -- 2 };
  a -- 1;
};
```

In the above example, there is no node before the double slash, which means that the two sublayouts will be part of the main graph, but will not be indicated otherwise.



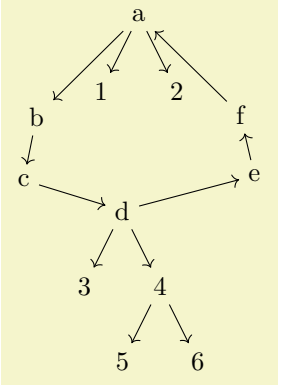

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular,trees}
\tikz \graph [simple necklace layout] {
 // [simple necklace layout] { a -> b -> c -> d -> e -> f -> a };

 // [tree layout] { % first tentacle
   a -> {1, 2};
 };

 // [tree layout] {% second tentacle
   d -> {3, 4 -> {5, 6}}
 };
};
```

In the above example, the first sublayout is the one for the nodes with letter names. These nodes are arranged using a simple necklace layout as the sublayout inherits this option from the main layout. The two small trees (`a -> {1, 2}` and the tree starting at the `d` node) are also sublayouts, triggered by the `tree layout` option. They are also arranged. Then, all of the layouts are merged (as described later). The result is actually a single node, so the main layout does nothing here.

Compare the above to the following code:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular,trees}
\tikz \graph [simple necklace layout] {
  // [tree layout] { % first ``giant node''
    a -> {1, 2};
  };

  a -> b -> c -> d;

  // [tree layout] {% second ``giant node''
    d -> {3, 4 -> {5, 6}}
  },

  d -> e -> f -> a;
};
```

Here, only the two trees are laid out first. They are then contracted into "giant nodes" and these are then part of the set of nodes that are arranged by the `simple necklace layout`. For details of how this contracting works, see below.

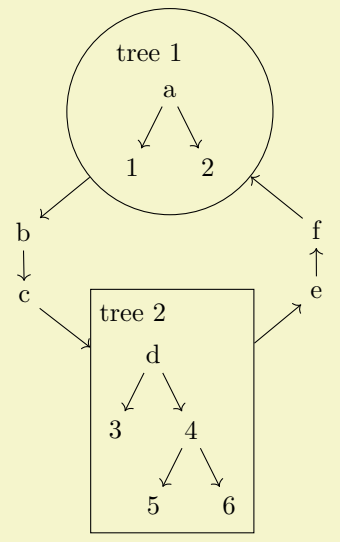
28.10.2 Subgraph Nodes

A *subgraph node* is a special kind of node that "surrounds" the vertices of a subgraph. The special property of a subgraph node opposed to a normal node is that it is created only after the subgraph has been laid out. However, the difference to a collection like `hyper` is that the node is available immediately as a normal node in the sense that you can connect edges to it.

The syntax used to declare a subgraph node in a `graph` specification is as follows:

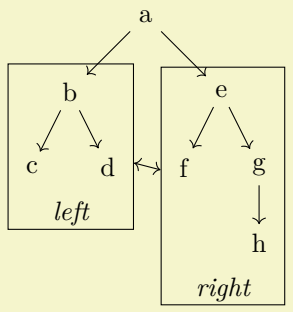
"⟨*node name*⟩"/"⟨*text*⟩" [⟨*node options*⟩] // [⟨*layout options*⟩] {⟨*subgraph*⟩}

The idea ist that a subgraph node is declared like a normal node specification, but is followed by a double slash and a subgraph:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular,trees}
\tikz \graph [simple necklace layout] {
     tree 1[draw, circle] // [tree layout] { a -> {1, 2}; }
  -> b
  -> c
  -> tree 2[draw] // [tree layout] { d -> {3, 4 -> {5, 6} } }
  -> e
  -> f
  -> tree 1;
};
```

Note how the two subgraph nodes `tree 1` and `tree 2` surround the two smaller trees. In the example, both had trees as contents and these trees were rendered using a sublayout. However, a subgraph layout does not need to have its own layout: If you do *not* provide a layout name after the double slash, the subgraph node will simply surround all nodes that were placed by the main layout wherever they were placed:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [subgraph text bottom=text centered,
       subgraph nodes={font=\itshape}]
  \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    left [draw]  // { b, c, d };
    right [draw] // { e, f, g, h};

    left <-> right;
  };
```

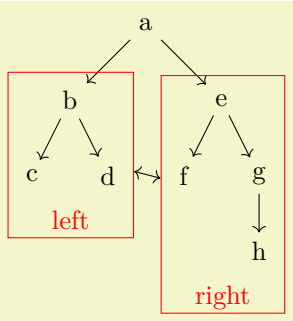
Every time a subgraph node is created, the following style is execute:

`/tikz/every subgraph node` (no value)

Set a subgraph node style.

`/tikz/subgraph nodes`=⟨*style*⟩ (no default)

Sets the `every subgraph node` style to ⟨*style*⟩.



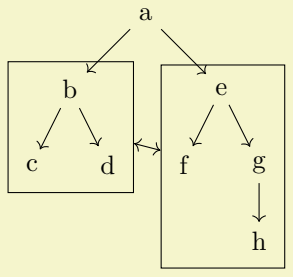

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [subgraph text bottom=text centered,
       subgraph nodes=red]
  \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    left [draw]  // { b, c, d };
    right [draw] // { e, f, g, h};

    left <-> right;
  };
```

`/tikz/subgraph text none` (no value)

When this option is used, the text of a subgraph node is not shown. Adding a slash after the node name achieves roughly the same effect, but this option is useful in situations when subgraph nodes generally should not have any text inside them.



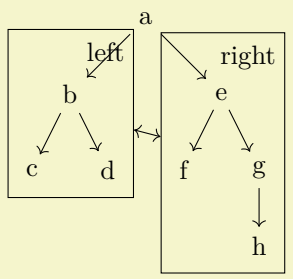

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [subgraph text none]
  \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    left [draw]  // { b, c, d };
    right [draw] // { e, f, g, h};

    left <-> right;
  };
```

`/tikz/subgraph text top=`⟨*text alignment options*⟩ (default `text ragged right`)

Specifies that the text of a subgraph node should be placed at the top of the subgraph node: Still inside the node, but above all nodes inside the subgraph node.

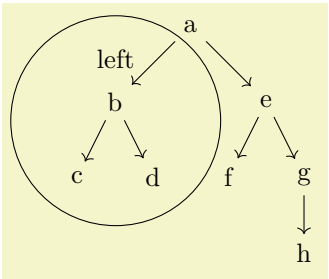



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [subgraph text top=text ragged left]
  \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    left [draw]  // { b, c, d };
    right [draw] // { e, f, g, h};

    left <-> right;
  };
```

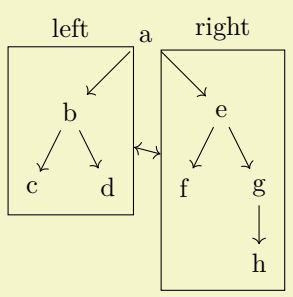
You can pass any of the ⟨*text alignment options*⟩ understood by Ti*k*Z, such as `text centered`:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [subgraph text top=text centered]
  \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    left [draw, circle] // { b, c, d };
  };
```

To place a label *outside* the subgraph node, use a label, typically defined using the `quotes` library:




```
\usetikzlibrary {graphs,graphdrawing,quotes} \usegdlibrary {trees}
\tikz \graph [tree layout] {
    a -> { b -> {c, d}, e -> {f, g -> h} };

    / ["left", draw]  // { b, c, d } <->
    / ["right", draw] // { e, f, g, h};
  };
```

`/tikz/subgraph text bottom=`⟨*text alignment options*⟩ (default `ragged right`)

Works like `subgraph text top`, only the text placed at the bottom.

Note that there are no keys `subgraph text left` or `... right`, for somewhat technical reasons.

`/tikz/subgraph text sep=`⟨*dimension*⟩ (no default, initially `.1em`)

Some space added between the inner nodes of a subgraph node and the text labels.

28.10.3 Overlapping Sublayouts

Nodes and edges can be part of several layouts. This will inevitably lead to conflicts because algorithm will disagree on where a node should be placed on the canvas. For this reason, there are some rules governing how such conflicts are resolved: Given a layout, starting with the main layout, the graph drawing system does the following:

1. We start by first processing the (direct) sublayouts of the current layout (recursively). Sublayouts may overlap (they may share one or more nodes), but we run the specified layout algorithm for each sublayout independently on a "fresh copy" of all the nodes making up the sublayout. In particular, different, conflicting positions may be computed for nodes when they are present in several sublayouts.

2. Once all nodes in the sublayouts have been laid out in this way, we *join* overlapping elements. The idea is that if two layouts share exactly one vertex, we can shift them around so that his vertex is at the same position in both layouts. In more detail, the following happens:

   We build a (conceptual) graph whose nodes are the sublayouts and in which there is an edge between two nodes if the sublayouts represented by these elements have a node in common. Inside the resulting graph, we treat each connected component separately. Each component has the property that the sublayouts represented by the nodes in the component overlap by at least one node. We now *join* them as follows: We start with the first sublayout in the component ("first" with respect to the order in which they appear in the input graph) and "mark" this sublayout. We loop the following instructions as long as possible: Search for the first sublayout (again, with respect to the order in which they appear in the input) that is connect by an edge to a marked sublayout. The sublayout will now have at least one node in common with the marked sublayouts (possibly, even more). We consider the first such node (again, first respect to the input ordering) and shift the whole sublayout is such a way that this particular node is at the position is has in the marked sublayouts. Note that after the shift, other nodes that are also present in the marked sublayouts may lie at a different position in the current sublayout. In this case, the position in the marked sublayouts "wins". We then mark the sublayout.

3. When the above algorithm has run, we will have computed positions for all nodes in all sublayouts of each of the components. For each component, we contract all nodes of the component to a single node. This new node will be "large" in the sense that its convex hull is the convex hull of all the nodes in the component. All nodes that used to be part of the component are removed and the new large node is added (with arcs adjusted appropriately).

4. We now run the layout's algorithm on the resulting nodes (the remaining original nodes and the contracted nodes).

5. In a last step, once the graph has been laid out, we expand the nodes that were previously contracted. For this, the nodes that were deleted earlier get reinserted, but shifted by whatever amount the contraction node got shifted.

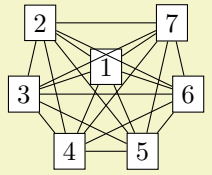
28.11 Miscellaneous Options

`/graph drawing/nodes behind edges=`⟨*boolean*⟩ (default `true`)

Specifies, that nodes should be drawn behind the edges Once a graph drawing algorithm has determined positions for the nodes, they are drawn *before* the edges are drawn; after all, it is hard to draw an edge between nodes when their positions are not yet known. However, we typically want the nodes to be rendered *after* or rather *on top* of the edges. For this reason, the default behavior is that the nodes at their final positions are collected in a box that is inserted into the output stream only after the edges have been drawn – which has the effect that the nodes will be placed "on top" of the edges.

This behavior can be changed using this option. When the key is invoked, nodes are placed *behind* the edges.

Example

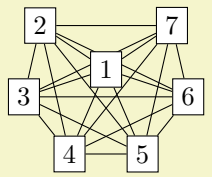



```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {circular,examples}
\tikz \graph [simple necklace layout, nodes={draw,fill=white},
              nodes behind edges]
  { subgraph K_n [n=7], 1 [regardless at={(0,-1)}] };
```

`/graph drawing/edges behind nodes=`⟨*string*⟩

This is the default placement of edges: Behind the nodes.

Example




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {circular,examples}
\tikz \graph [simple necklace layout, nodes={draw,fill=white},
              edges behind nodes]
  { subgraph K_n [n=7], 1 [regardless at={(0,-1)}] };
```

`/graph drawing/random seed=`⟨*number*⟩ (initially `42`)

To ensure that the same is always shown in the same way when the same algorithm is applied, the random is seed is reset on each call of the graph drawing engine. To (possibly) get different results on different runs, change this value.

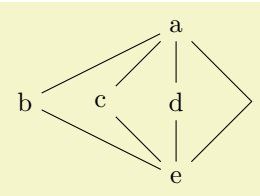
`/graph drawing/variation=`⟨*number*⟩

An alias for `random seed`.

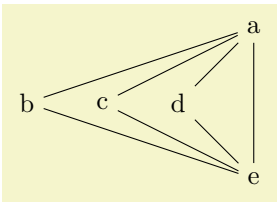
`/graph drawing/weight=`⟨*number*⟩ (initially `1`)

Sets the "weight" of an edge or a node. For many algorithms, this number tells the algorithm how "important" the edge or node is. For instance, in a `layered layout`, an edge with a large `weight` will be as short as possible.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout] {
  a -- {b,c,d} -- e -- a;
};
```

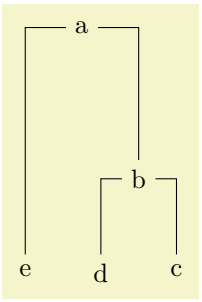



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout] {
  a -- {b,c,d} -- e --[weight=3] a;
};
```

`/graph drawing/length=`⟨*length*⟩ (initially `1`)

Sets the "length" of an edge. Algorithms may take this value into account when drawing a graph.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics,trees}
\tikz \graph [phylogenetic tree layout] {
  a --[length=2] b --[length=1] {c,d};
  a --[length=3] e
};
```

`/graph drawing/radius=`⟨*number*⟩ (initially `0`)

The radius of a circular object used in graph drawing.

`/graph drawing/no layout=`⟨*string*⟩

This layout does nothing.

29 Using Graph Drawing in PGF

by Till Tantau

PGF Library graphdrawing

```
\usepgflibrary{graphdrawing}  % LaTeX and plain TeX
\usepgflibrary[graphdrawing]  % ConTeXt
```

This package provides the core support for graph drawing inside PGF. It does so by providing PGF macros for controlling the graph drawing system, but also implements the binding to the graph drawing system (see Section 39 for details on bindings).

29.1 Overview

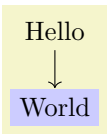
Just like everywhere else in PGF, Ti*k*Z is "just a convenient syntax" in the context of graph drawing. The "hard work" of binding the internal representations of nodes and edges with the graph drawing system written in Lua is not done by Ti*k*Z, but rather by a set of macros that are part of the basic PGF layer.

The documentation of the PGF part of the graph drawing system that is presented in the following includes only those macros that other TEX packages could conceivably call in order to use the graph drawing system without using Ti*k*Z; for instance, for efficiency reasons. (The internal callback functions defined in the `graphdrawing` library that are part of the binding between PGF and the graph drawing system are not documented, should not be called, and may change in the future.)

29.2 How Graph Drawing in PGF Works

The core idea behind graph drawing in PGF is that inside special *graph drawing scopes* whenever PGF creates a node, we intercept this node creation and *do not* immediately place the node. Rather, we pass it down to Lua part of the graph drawing system via calls to appropriate methods of the (Lua) class `InterfaceToDisplay`. The effect will be that the nodes are "tucked away" in some internal tables. For edges, we introduce a special command called `\pgfgdedge` that tells the graph drawing system that there is an edge between two tucked-away nodes. Then, at the end of the graph drawing scope, a graph drawing algorithm written in Lua starts to work on the graph by computing new positions for the nodes. Once the algorithm has finished, the graph drawing system starts sending back the nodes and edges to PGF via the methods of the class `BindingToPGF`. These methods reinsert some code into the TEX output stream that finally places the nodes at their final positions. Note that graph drawing algorithms are perfectly oblivious to all of this; indeed, the graph drawing algorithms can even be used independently of TEX.

Let us have a look at a simple example to see what happens when a graph is specified:



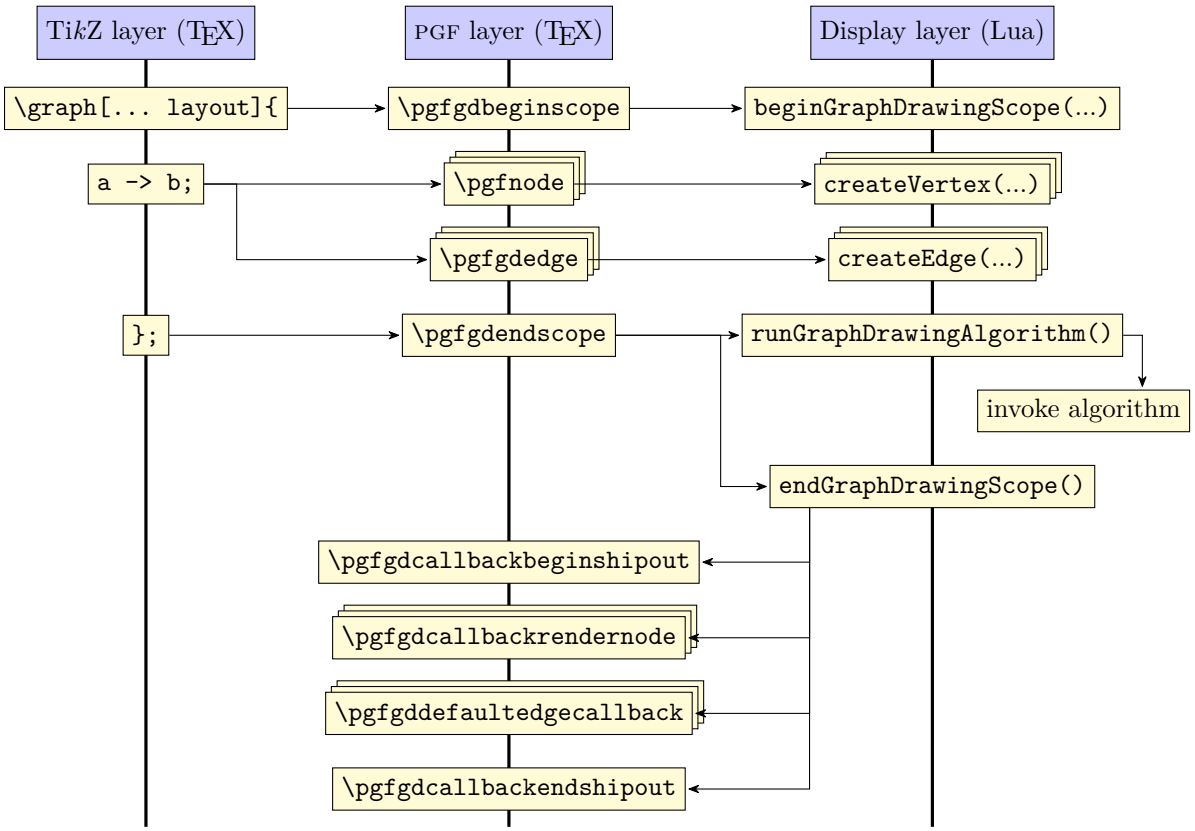

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz[tree layout]
  \graph {root [as=Hello] -> World[fill=blue!20]};
```

The key `tree layout` internally calls the key `request scope and layout`, which in turn calls the macro `\pgfgdbeginscope`, which starts a graph drawing scope inside the graph drawing system. Once this macro has been called, until the next call of `\pgfgdendscope`, all nodes that are created actually get passed down to the graph drawing engine. This is implemented on the lowest layer, namely by directly intercepting nodes freshly created using `\pgfnode`. In our example, this happens in two places: For the `root` node and for the `World` node. The `graphs` library and Ti*k*Z internally call the `\pgfnode` macro for these two nodes (after a large number of internal syntax translations, but the graph drawing system does not care about them).

Note that the node boxes will have been fully created before they are passed down to the graph drawing engine – only their final position is not yet fixed. It is not possible to modify the size of nodes inside the graph drawing engine, but you can create new nodes in certain situations.

In contrast, the single edge of the graph that is created by the `->` command is not fully created before it is passed down to the graph drawing system. This would not really make sense since before the final positions of the nodes are fixed, we cannot even begin to compute the length of this edge, let alone where it should start or end. For this reason, on the upper Ti*k*Z layer, the normal edge creation that would be caused by `->` via `new ->` is suppressed. Instead, the command `\pgfgdedge` is called. Similarly, inside a graph drawing scope, Ti*k*Z will suppress both the `edge` and the `edge from parent` command and cause `\pgfgdedge` to be called instead.

An overview of what happens is illustrated by the following call graph:



The above diagram glosses over the fact that the display layer does not actually call any of the macros of TEX directly, but uses a so called *binding* (see the class `BindingToPGF`). However, this will not be important for the present section since you cannot access the binding directly.

29.2.1 Graph Drawing Scopes

When the graph drawing system is active, some pretty basic things inside PGF change – such as the fact that nodes are no longer created in the normal manner. For this reason, the graph drawing system must be switched on and of explicitly through opening and closing a so called *graph drawing scope*. These scopes can, in principle, be nested, namely a graph contains a node that contains some text that in turn contains a subpicture that contains a drawing of a graph. However, this is *not* the same as subgraphs nodes and sublayouts, which are all part of the same graph drawing scope. Normally, graph drawing scopes are not nested.

Graph drawing scopes are created using the following commands:

`\pgfgdbeginscope`

This macro starts a TEX scope inside which the following things happen:

1. The display layer method `beginGraphDrawingScope` is called, which created a new graph drawing scope inside the graph drawing system and places it on top of an internal stack. From now on, all subsequent interface calls will refer to this scope until `\pgfgdendscope` is called, which will pop the scope once more.
2. Inside the TEX scope, nodes are not placed immediately. Rather, `\pgfpositionnodelater`, see Section 106.2.3, is used to call `InterfaceToDisplay.createVertex` for all nodes created inside the scope. This will cause them to be put inside some internal table.
3. Some additional ⟨*code*⟩ is executed, which has been set using the following command:

   `\pgfgdaddspecificationhook{`⟨*code*⟩`}`

   This command adds the ⟨*code*⟩ to the code that is executed whenever a graph drawing scope starts. For instance, the TikZ library `graphdrawing` uses this macro to add some ⟨*code*⟩ that will redirect the `edge` and `edge from parent` path commands to `\pgfgdedge`.

4. `\pgftransformreset` is called.
5. The following TEX-if is set to true:

   `\ifpgfgdgraphdrawingscopeactive`

   Will be true inside a graph drawing scope.

The above has a number of consequences for what can happen inside a graph drawing scope:

- Since nodes are not actually created before the end of the scope, you cannot reference these nodes. Thus, you cannot write

```
\tikz [spring layout] {
  \node (a) {a};
  \node (b) {b};
  \draw (a) -- (b);
}
```

  The problem is that we cannot connect `(a)` and `(b)` via a straight line since these nodes do not exist at that point (they are available only deeply inside the Lua).
- In order to create edges between nodes inside a graph drawing scope, you need to call the `\pgfgdedge` command, described below.

Additionally, when Ti*k*Z is used, the following things also happen:

- If the `graphs` library has been loaded, the default positioning mechanisms of this library are switched off, leaving the positioning to the graph drawing engine. Also, when an edge is created by the `graphs` library, this is signalled to the `graphdrawing` library. (To be more precise: The keys `new ->` and so on are redefined so that they call `\pgfgdedge` instead of creating an edge.
- The `edge` path command is modified so that it also calls `\pgfgdedge` instead of immediately creating any edges.
- The `edge from parent` path command is modified so that is also calls `\pgfgdedge`.
- The keys `append after command` and `prefix after command` keys are modified so that they are executed only via `late options` when the node has "reached its final parking position".

Note that inside a graph drawing scope you first have to open a (main) layout scope (using the `\pgfgdbeginlayout` command described later on) before you can add nodes and edges to the scope.

`\pgfgdendscope`

This macro is used to end a graph drawing scope. It must be given on the same TEX grouping level as the corresponding `\pgfgdbeginscope`. When the macro is called, it triggers a lot of new calls:

1. The special treatment of newly created boxes is ended. Nodes are once more created normally.
2. The effects of the ⟨*code*⟩ that was inserted via the specification hook command also ends (provided it had no global effects).
3. We call `InterfaceToDisplay.runGraphDrawingAlgorithm`. This will cause the algorithm(s) for the graph to be executed (since a graph can have sublayouts, several algorithms may be run). See Section 29.3 below.
4. Next, we call `InterfaceToDisplay.endGraphDrawingScope`. This causes all nodes that were intercepted during the graph drawing scope to be reinserted into the output stream at the positions that were computed for them. Also, for each edge that was requested via `\pgfgdedge`, the callback macro is called (see below).

Inside a graph drawing scope, nodes are automatically passed down to the graph drawing engine, while for edges a command has to be called explicitly:

`\pgfgdedge{`⟨*first node*⟩`}{`⟨*second node*⟩`}{`⟨*edge direction*⟩`}{`⟨*edge options*⟩`}{`⟨*edge nodes*⟩`}`

This command is used to tell the graph drawing engine that there is an edge between ⟨*first node*⟩ and ⟨*second node*⟩ in your graph. The "kind" of connection is indicated by ⟨*direction*⟩, which may be one of the following:

- `->` indicates a directed edge (also known as an arc) from ⟨*first node*⟩ to ⟨*second node*⟩.

- `--` indicates an undirected edge between ⟨*first node*⟩ and ⟨*second node*⟩,
- `<-` indicates a directed edge from ⟨*second node*⟩ to ⟨*first node*⟩, but with the "additional hint" that this is a "backward" edge. A graph drawing algorithm may or may not take this hint into account.
- `<->` indicates a bi-directed edge between ⟨*first node*⟩ and ⟨*second node*⟩.
- `-!-` indicates that the edge from ⟨*first node*⟩ to ⟨*second node*⟩ is "missing".

Note that in all cases, the syntactic digraph will contain an arc from ⟨*first node*⟩ to ⟨*second node*⟩, regardless of the value of ⟨*direction*⟩. The ⟨*direction*⟩ is "just" a "semantic annotation".

The parameters ⟨*edge options*⟩ and ⟨*edge nodes*⟩ are a bit more tricky. When an edge between two vertices of a graph is created via `\pgfgdedge`, nothing is actually done immediately. After all, without knowing the final positions of the nodes ⟨*first node*⟩ and ⟨*second node*⟩, there is no way of creating the actual drawing commands for the edge. Thus, the actual drawing of the edge is done only when the graph drawing algorithm is done (namely in the macro `\pgfgdedgecallback`, see later).

Because of this "delayed" drawing of edges, options that influence the edge must be retained until the moment when the edge is actually drawn. Parameters ⟨*edge options*⟩ and ⟨*edge nodes*⟩ store such options.

Let us start with ⟨*edge options*⟩. This parameter should be set to a list of key–value pairs like

```
/tikz/.cd, color=red, very thick, orient=down
```

Some of these options may be of interest to the graph drawing algorithm (like the last option) while others will only be important during the drawing of edge (like the first option). The options that are important for the graph drawing algorithm must be pushed onto the graph drawing system's option stack.

The tricky part is that options that are of interest to the graph drawing algorithm must be executed *before* the algorithm starts, but the options as a whole are usually only executed during the drawing of the edges, which is *after* the algorithm has finished. To overcome this problem, the following happens:

The options in ⟨*edge options*⟩ are executed "tentatively" inside `\pgfgdedge`. However, this execution is done in a "heavily guarded sandbox" where all effects of the options (like changing the color or the line width) do not propagate beyond the sandbox. Only the changes of the graph drawing edge parameters leave the sandbox. These parameters are then passed down to the graph drawing system.

Later, when the edge is drawn using `\pgfgdedgecallback`, the options ⟨*edge options*⟩ are available once more and then they are executed normally.

Note that when the options in ⟨*edge options*⟩ are executed, no path is preset. Thus, you typically need to start it with, say, `/tikz/.cd`. Also note that the sandbox is not perfect and changing global values will have an effect outside the sandbox. Indeed, "putting things in a sandbox" just means that the options are executed inside a TEX scope inside an interrupted path inside a TEX box that is thrown away immediately.

The text in ⟨*edge nodes*⟩ is some "auxiliary" text that is simply stored away and later directed to `\pgfgdedgecallback`. This is used for instance by Ti*k*Z to store its node labels.

`\pgfgdsetedgecallback{`⟨*macro*⟩`}`

This command allows you to change the ⟨*macro*⟩ that gets called form inside the graph drawing system at the end of the creation of a graph, when the nodes have been positioned. The ⟨*macro*⟩ will be called once for each edge with the following parameters:

⟨*macro*⟩`{`⟨*first node*⟩`}{`⟨*second node*⟩`}{`⟨*direction*⟩`}{`⟨*edge options*⟩`}{`⟨*edge nodes*⟩`}`
`{`⟨*algorithm-generated options*⟩`}{`⟨*bend information*⟩`}{`⟨*animations*⟩`}`

The first five parameters are the original values that were passed down to the `\pgfgdedge` command.

The ⟨*algorithm-generated options*⟩ have been "computed by the algorithm". For instance, an algorithm might have determined, say, flow capacities for edges and it might now wish to communicate this information back to the upper layers. These options should be executed with the path `/graph drawing`.

The parameter ⟨*bend information*⟩ contains algorithmically-computed information concerning how the edge should bend. This will be a text like `(10pt,20pt)--(30pt,40pt)` in Ti*k*Z-syntax and may include the path commands `--`, `..` (followed by Bézier coordinates), and `--cycle`.

The last parameter ⟨*animations*⟩ contains algorithmically-generated animation commands (calls to `\pgfanimateattribute`). The `whom` will be set to `pgf@gd`.

The default ⟨*macro*⟩ simply draws a line between the nodes. When the `graphdrawing` library of the Ti*k*Z layer is loaded, a more fancy ⟨*macro*⟩ is used that takes all of the parameters into account.

29.3 Layout Scopes

As described in Section 28.10, the graph drawing engine does not always apply only a single algorithm. Rather, several different algorithm may be applied to different parts of the graph. How this happens, exactly, is governed by a hierarchy of layouts, which are setup using the commands `\pgfgdbeginlayout` and `\pgfgdendlayout`.

`\pgfgdbeginlayout`

This command first starts a new TEX scope and then informs the display layer that a new (sub)layout should be started. For each graph there may be a hierarchy of layouts, each of which contains a certain number of vertices and edges. This hierarchy is created through calls to this macros and the corresponding calls of `\pgfgdendlayout`. For each graph drawing scope there has to be exactly one main layout that encompasses all nodes and edges and also all sublayouts. Thus, after a graph drawing scope has been opened, a layout scope also needs to be opened almost immediately.

For each layout created via this macro, a graph drawing algorithm will be run later on the subgraph of all nodes that make up the layout. Which algorithm is run for the layout is dictated by which layout key (one of the `... layout` keys) is "in force" when the macro is called. Thus, using a layout key for selecting an algorithm must always be done *before* the layout is started. (However, see the discussion of layout keys in the next subsection for more details on what really happens.)

A vertex can be part of several layouts, either because they are nested or because they overlap (this happens when a node is later on added to another layout by calling `\pgfgdsetlatenodeoption`). This means that it is not immediately obvious how conflicts arising from the different ways different algorithms "would like to place nodes" should be resolved. The method for this resolving is detailed in Section 28.10.3.

`\pgfgdendlayout`

This command ends the TEX scope of the current layout. Once closed, no nodes or edges can be added to a layout.

`\pgfgdsetlatenodeoption{`⟨*node name*⟩`}`

This command can only be called when the node named ⟨*node name*⟩ has already been created inside the current graph drawing scope. The effect of calling this macro will be that all options currently on the graph drawing system's option stack will be added to the node's option, possibly overwriting the original option settings. Furthermore, the node will become part of all layouts currently on the option stack. This means that you can use this command to add a node to several layouts that are not included in one another.

29.4 Layout Keys

Layout keys are keys like `tree layout` or `layered layout` that are used to select a specific graph drawing algorithm. From the graph drawing system's point of view, these keys "just" select an algorithm and when several layout keys are used in a row, the last one would "win"; just as when you say `orient=90` directly followed by `orient=0`, the result is that the `orient` key is set to `0` because the last key "wins".

Unfortunately, if keys like `tree layout` were "just" to select an algorithm, we would still need a key or some special syntax to actually start a (sub)layout. In early versions of the system this was exactly what people had to do and this was somewhat awkward. Because of this problem, the behavior of the layout keys in PGF (and only there, other display layers need to implement their own behavior) is now a bit more involved. When you use a key like `tree layout` (more precisely, any key that was declared as an algorithm key on the algorithm layer of the graph drawing system) in any scope in PGF, the following happens:

1. The graph drawing system is told that a specific algorithm has been selected (the Reingold–Tilford-algorithm in this case; this information was communicated to the graph drawing system during the declaration of the algorithm). Being "told" about this means that a special entry is pushed onto the current options stack of the graph drawing system.

2. An internal “request” for a “scope and a layout” is made. This has several effects:
3. We first test whether we are already inside a layout scope. If not, we use `\pgfgdbeginscope` to open a graph drawing scope. This scope will be closed appropriately (see `\pgfgdsetrequestcallback` for details).
4. Next, a layout scope is opened using `\pgfgdbeginlayout`. It will also be closed appropriately.

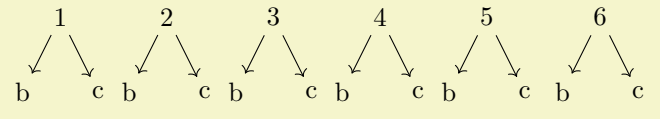
The net effect of the above is that the first use of a layout key in a picture starts both a graph drawing scope and also a main layout, while subsequent uses of layout keys inside a picture will only open sublayouts.

`\pgfgdsetrequestcallback{`⟨*macro*⟩`}`

This command sets up ⟨*macro*⟩ as the macro that is called whenever a layout key “requests” that a layout and, possibly, a graph drawing scope is opened. When ⟨*macro*⟩ is called, it gets two parameters, the ⟨*begin code*⟩ and the ⟨*end code*⟩. In addition to whatever setup the ⟨*macro*⟩ would like to do, it should execute the ⟨*begin code*⟩ at the beginning of a TEX scope (the code will open graph drawing and layout scopes) and the ⟨*end code*⟩ at the end of the same TEX scope.

The need for this slightly strange macro arises from the fact that in Ti*k*Z we often write things like `[spring layout,node sep=2cm]`. The point is that when the `spring layout` key is executed, we do *not* wish to open a layout scope immediately. Rather, this should happen only after the option `nodes sep=2cm` has been executed. For this reason, Ti*k*Z sets up a special ⟨*macro*⟩ that “delays” the execution of the ⟨*begin code*⟩ until the end of the opening of the next scope.

Because of this, in Ti*k*Z layout keys can only be used as an option when a Ti*k*Z scope is started. Thus, you can pass them to `\tikz`, to `{tikzpicture}`, to `\scoped`, to `{scope}`, to `graph`, and to `{graph}`. For instance, the `tree layout` option can be used in the following ways:



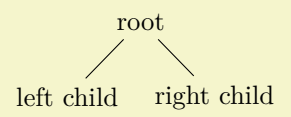

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [tree layout] \graph          {1 -> {b,c}};
\tikz \graph [tree layout]          {2 -> {b,c}};
\tikz \path graph [tree layout]     {3 -> {b,c}};

\begin{tikzpicture}[tree layout]
  \graph                            {4 -> {b,c}};
\end{tikzpicture}

\begin{tikzpicture}
  \scoped [tree layout] \graph      {5 -> {b,c}};

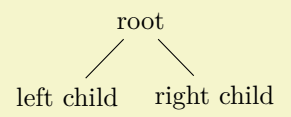
  \begin{scope}[tree layout, xshift=1.5cm]
    \graph                          {6 -> {b,c}};
  \end{scope}
\end{tikzpicture}
```

You can *not* use layout keys with a single node or on a path. In particular, to typeset a tree given in the `child` syntax somewhere inside a `{tikzpicture}`, you must prefix it with the `\scoped` command:



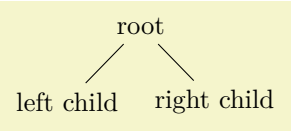

```
\usetikzlibrary {graphdrawing} \usegdlibrary {trees}
\begin{tikzpicture}
  \scoped [tree layout]
    \node {root}
    child { node {left child} }
    child { node {right child} };
\end{tikzpicture}
```

Naturally, the above could have been written more succinctly as




```
\usetikzlibrary {graphdrawing} \usegdlibrary {trees}
\tikz [tree layout]
  \node {root}
  child { node {left child} }
  child { node {right child} };
```

Or even more succinctly:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout] { root -- {left child, right child} };
```

29.5 Parameters

When a graph drawing algorithm starts working, a set of options, called "graph drawing parameters" or just "parameters" can influence the way the algorithm works. For instance, a graph drawing parameter might be the average distance between vertices which the algorithm should take into account. Another example might be the fact the certain nodes are special nodes and that a certain edge should have a large label.

These graph drawing parameters are different from "usual" PGF options: An algorithmic parameter influences the way the algorithm works, while usual options influence the way the result looks like. For instance, the fact that a node is red is not a graph drawing parameter, while the shape of a node might be an graph drawing parameter.

The possible graph parameters are declared by the algorithmic layer through the `declare` method; you cannot declare parameters on the PGF layer since this would not be compatible across different display systems.

Users use a graph parameter in the same way as a normal key. The difference is that each time a key representing a graph drawing parameter is used, a special function of the graph drawing system's interface is called to "push" the parameter onto an internal option stack (and elements are popped from this stack whenever the TEX scope closes in which the key was used).

The net effect of all of this is that the graph drawing system keeps track of a stack of option in parallel to TEX. You cannot, however, access the current values of graph drawing parameters from TEX since they are tucked away deep inside the graph drawing system.

29.6 Events

Events are used to pass information from the parser about the syntactic structure of a graph to graph drawing algorithms. Consider, for instance, a graph that is actually a tree in which some node "misses" its first child. In this case, the information that the child is missing is neither part of any node (because the node is missing, after all) nor is it an option of the whole graph. However, events are created by the parser to allow an algorithm to reconstruct the fact that the child is missing. Naturally, graph drawing algorithms may choose to ignore events and most will.

Most of the creation and handling of events is done automatically. The only reason you might wish to use the following commands is when you write a "parser extension" together with a new graph drawing algorithm. For instance, you might come up with new options that, when used, trigger events.

`\pgfgdevent{`⟨*kind*⟩`}{`⟨*parameter*⟩`}`

Calls `createEvent` of the graph drawing system's interface class. This creates a new `Event` object on the Lua layer whose `kind` field is set to ⟨*kind*⟩ and the `parameters` field to ⟨*parameter*⟩. You must be inside a graph drawing scope to use this command.

`\pgfgdbegineventgroup{`⟨*parameter*⟩`}`

Starts an event group. This just means that an `Event` of kind `begin` is created with the given ⟨*parameter*⟩.

`\pgfgdendeventgroup`

Ends an event group. This is done by adding an event of kind `end` without any parameters to the event string.

`\pgfgdeventgroup{`⟨*parameters*⟩`}`

Starts an event group just like `\pgfgdbegineventgroup`, but adds a corresponding closing `end` event at the end of the current TEX group (using `\aftergroup`).

29.7 Subgraph Nodes

\pgfgdsubgraphnode{⟨*name*⟩}{⟨*node options*⟩}{⟨*node text*⟩}

A subgraph node is a node that "surrounds" the nodes of a subgraph. The special property of a subgraph node opposed to a normal node is that it is created only after the subgraph has been laid out. However, the difference to a collection like `hyper` is that the node is available immediately as a normal node in the sense that you can connect edges to it.

What happens internally is that subgraph nodes get "registered" immediately both on the PGF level and on the Lua level, but the actual node is only created inside the layout pipeline using a callback. The actual node creation happens when the innermost layout in which the subgraph node is declared has finished.

When you create a subgraph node using this macro, you also start a collection (of an internal kind) that stores the subgraph. All following nodes in the current TEX scope will become part of this collection.

The ⟨*name*⟩ is the node name by which you can refer to this node in the following. The ⟨*node options*⟩ are normal PGF options (like `red` or `draw` or `circle`) that will influence the appearance when it is created later on. The ⟨*node text*⟩ is the text that will be passed to `\pgfnode` upon creation of the node.

See `InterfaceToDisplay.pushSubgraphVertex` for more details.

30 Graph Drawing Layouts: Trees

by Till Tantau

Graph Drawing Library `trees`

```
\usegdlibrary{trees} % LaTeX and plain TeX
\usegdlibrary[trees] % ConTeXt
```

Ti*k*Z offers several different syntax to specify trees (see Sections 19 and 21). The job of the graph drawing algorithms from this library is to turn the specification of trees into beautiful layouts.

We start this section with a description of algorithms, then we have a look at how missing children can be specified and at what happens when the input graph is not a tree.

30.1 The Tree Layouts

30.1.1 The Reingold–Tilford Layout

`/graph drawing/tree layout=`⟨*string*⟩

This layout uses the Reingold–Tilform method for drawing trees. The Reingold–Tilford method is a standard method for drawing trees. It is described in:

- E. M. Reingold and J. S. Tilford, Tidier drawings of trees, *IEEE Transactions on Software Engineering,* 7(2), 223–228, 1981.

My implementation in `graphdrawing.trees` follows the following paper, which introduces some nice extensions of the basic algorithm:

- A. Brüggemann-Klein, D. Wood, Drawing trees nicely with TeX, *Electronic Publishing,* 2(2), 101–115, 1989.

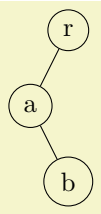
As a historical remark, Brüggemann-Klein and Wood have implemented their version of the Reingold–Tilford algorithm directly in TeX (resulting in the TreeTeX style). With the power of LuaTeX at our disposal, the 2012 implementation in the `graphdrawing.tree` library is somewhat more powerful and cleaner, but it really was an impressive achievement to implement this algorithm back in 1989 directly in TeX.

The basic idea of the Reingold–Tilford algorithm is to use the following rules to position the nodes of a tree (the following description assumes that the tree grows downwards, other growth directions are handled by the automatic orientation mechanisms of the graph drawing library):

1. For a node, recursively compute a layout for each of its children.
2. Place the tree rooted at the first child somewhere on the page.
3. Place the tree rooted at the second child to the right of the first one as near as possible so that no two nodes touch (and such that the `sibling sep` padding is not violated).
4. Repeat for all subsequent children.
5. Then place the root above the child trees at the middle position, that is, at the half-way point between the left-most and the right-most child of the node.

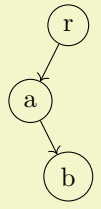
The standard keys `level distance`, `level sep`, `sibling distance`, and `sibling sep`, as well as the `pre` and `post` versions of these keys, as taken into consideration when nodes are positioned. See also Section 28.3 for details on these keys.

Handling of Missing Children. As described in Section 30.2, you can specify that some child nodes are "missing" in the tree, but some space should be reserved for them. This is exactly what happens: When the subtrees of the children of a node are arranged, each position with a missing child is treated as if a zero-width, zero-height subtree were present at that positions:



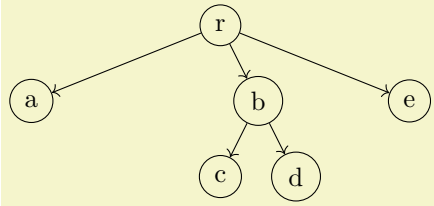

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [tree layout, nodes={draw,circle}]
  \node {r}
    child { node {a}
      child [missing]
      child { node {b} }
    }
    child[missing];
```

or in `graph` syntax:



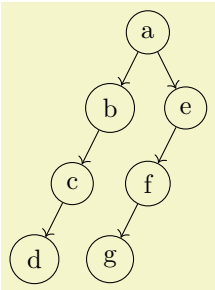

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
  \tikz \graph [tree layout, nodes={draw,circle}]
  {
    r -> {
      a -> {
        , %missing
        b},
      % missing
    }
  };
```

More than one child can go missing:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw,circle}, sibling sep=0pt]
  { r -> { a, , ,b -> {c,d}, ,e} };
```

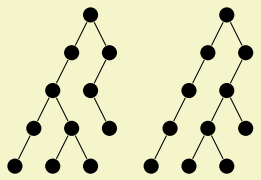
Although missing children are taken into consideration for the computation of the placement of the children of a root node relative to one another and also for the computation of the position of the root node, they are usually *not* considered as part of the "outline" of a subtree (the `minimum number of children` key ensures that `b`, `c`, `e`, and `f` all have a missing right child):




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, minimum number of children=2,
              nodes={draw,circle}]
  { a -> { b -> c -> d, e -> f -> g } };
```

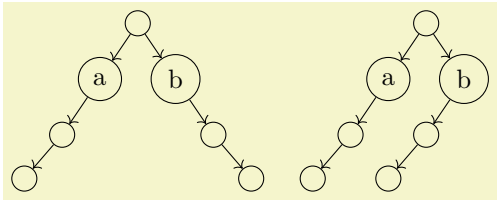
This behaviour of "ignoring" missing children in later stages of the recursion can be changed using the key `missing nodes get space`.

Significant Pairs of Siblings. Brüggemann-Klein and Wood have proposed an extension of the Reingold–Tilford method that is intended to better highlight the overall structure of a tree. Consider the following two trees:



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [baseline=(a.base), tree layout, minimum number of children=2,
       sibling distance=5mm, level distance=5mm]
  \graph [nodes={circle, inner sep=0pt, minimum size=2mm, fill, as=}]{
    a -- { b -- c -- { d -- e, f -- { g, h }}, i -- j -- k[second] }
  };\quad
\tikz [baseline=(a.base), tree layout, minimum number of children=2,
       sibling distance=5mm, level distance=5mm]
  \graph [nodes={circle, inner sep=0pt, minimum size=2mm, fill, as=}]{
    a -- { b -- c -- d -- e, i -- j -- { f -- {g,h}, k } }
  };
```

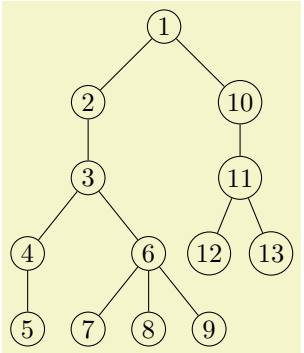
As observed by Brüggemann-Klein and Wood, the two trees are structurally quite different, but the Reingold–Tilford method places the nodes at exactly the same positions and only one edge "switches" positions. In order to better highlight the differences between the trees, they propose to add a little extra separation between siblings that form a *significant pair*. They define such a pair as follows: Consider the subtrees of two adjacent siblings. There will be one or more levels where these subtrees have a minimum distance. For instance, the following two trees the subtrees of the nodes `a` and `b` have a minimum distance only at the top level in the left example, and in all levels in the second example. A *significant pair* is a pair of siblings where the minimum distance is encountered on any level other than the first level. Thus, in the first example there is no significant pair, while in the second example `a` and `b` form such a pair.



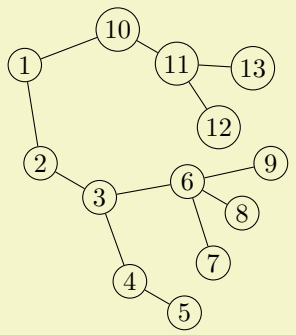

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, minimum number of children=2,
              level distance=5mm, nodes={circle,draw}]
  { / -> { a -> / -> /, b -> /[second] -> /[second] }};
  \quad
\tikz \graph [tree layout, minimum number of children=2,
              level distance=5mm, nodes={circle,draw}]
  { / -> { a -> / -> /, b -> / -> / }};
```

Whenever the algorithm encounters a significant pair, it adds extra space between the siblings as specified by the `significant sep` key.

Examples




```
\tikz [tree layout, sibling distance=8mm]
  \graph [nodes={circle, draw, inner sep=1.5pt}]{
    1 -- { 2 -- 3 -- { 4 -- 5, 6 -- { 7, 8, 9 }}, 10 -- 11 -- { 12, 13 } }
  };
```

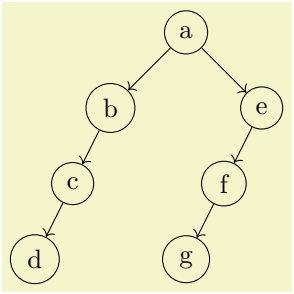



```
\tikz [tree layout, grow=-30,
       sibling distance=0mm, level distance=0mm,]
  \graph [nodes={circle, draw, inner sep=1.5pt}]{
    1 -- { 2 -- 3 -- { 4 -- 5, 6 -- { 7, 8, 9 }}, 10 -- 11 -- { 12, 13 } }
  };
```

/graph drawing/missing nodes get space=⟨*boolean*⟩ (default `true`)

When set to true, missing children are treated as if they where zero-width, zero-height nodes during the whole tree layout process.

Example

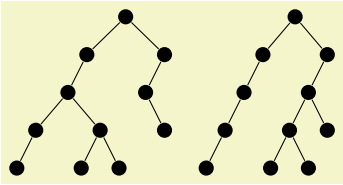



```
\tikz \graph [tree layout, missing nodes get space,
              minimum number of children=2, nodes={draw,circle}]
{ a -> { b -> c -> d, e -> f -> g } };
```

/graph drawing/significant sep=⟨*length*⟩ (initially `0`)

This space is added to significant pairs by the modified Reingold–Tilford algorithm.

Example



```
\tikz [baseline=(a.base), tree layout, significant sep=1em,
       minimum number of children=2,
       sibling distance=5mm, level distance=5mm]
  \graph [nodes={circle, inner sep=0pt, minimum size=2mm, fill, as=}]{
    a -- { b -- c -- { d -- e, f -- { g, h }}, i -- j -- k[second] }
  };\quad
\tikz [baseline=(a.base), tree layout, significant sep=1em,
       minimum number of children=2,
       sibling distance=5mm, level distance=5mm]
  \graph [nodes={circle, inner sep=0pt, minimum size=2mm, fill, as=}]{
    a -- { b -- c -- d -- e, i -- j -- { f -- {g,h}, k } }
  };
```

/graph drawing/binary tree layout=⟨*string*⟩

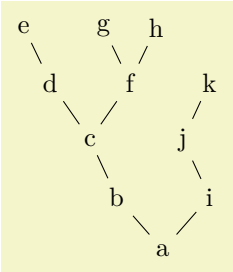
A layout based on the Reingold–Tilford method for drawing binary trees. This key executes:

1. `tree layout`, thereby selecting the Reingold–Tilford method,
2. `minimum number of children=2`, thereby ensuring the all nodes have (at least) two children or none at all, and

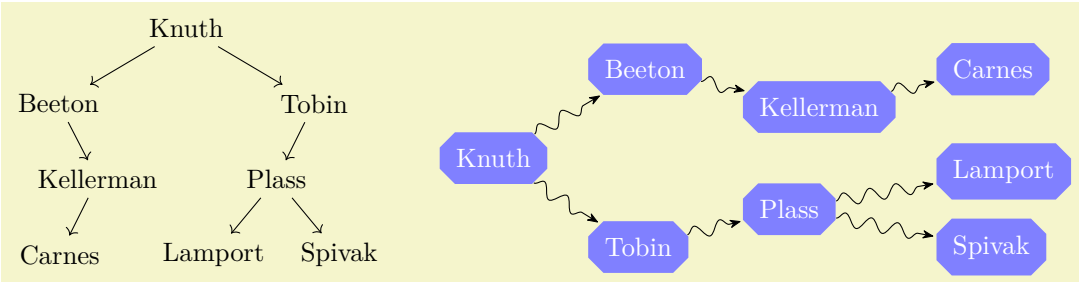
3. `significant sep=10pt` to highlight significant pairs.

In the examples, the last one is taken from the paper of Brüggemann-Klein and Wood. It demonstrates nicely the advantages of having the full power of Ti*k*Z's anchoring and the graph drawing engine's orientation mechanisms at one's disposal.

Examples




```
\tikz [grow'=up, binary tree layout, sibling distance=7mm, level distance=7mm]
  \graph {
    a -- { b -- c -- { d -- e, f -- { g, h }}, i -- j -- k[second] }
  };
```

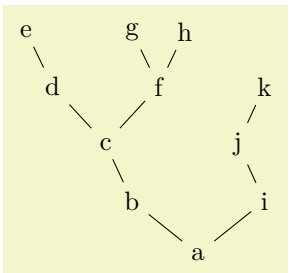



```
\usetikzlibrary {shapes.misc,arrows.meta,graphs,graphdrawing,decorations.pathmorphing} \usegdlibrary {trees}
\tikz \graph [binary tree layout] {
  Knuth -> {
    Beeton -> Kellerman [second] -> Carnes,
    Tobin -> Plass -> { Lamport, Spivak }
  }
};\qquad
\tikz [>={Stealth[round,sep]}]
  \graph [binary tree layout, grow'=right, level sep=1.5em,
          nodes={right, fill=blue!50, text=white, chamfered rectangle},
                edges={decorate,decoration={snake, post length=5pt}}]
  {
    Knuth -> {
      Beeton -> Kellerman [second] -> Carnes,
      Tobin -> Plass -> { Lamport, Spivak }
    }
  };
```

`/graph drawing/extended binary tree layout=`⟨*string*⟩

This algorithm is similar to `binary tree layout`, only the option `missing nodes get space` is executed and the `significant sep` is zero.

Example




```
\tikz [grow'=up, extended binary tree layout,
       sibling distance=7mm, level distance=7mm]
  \graph {
    a -- { b -- c -- { d -- e, f -- { g, h }}, i -- j -- k[second] }
  };
```

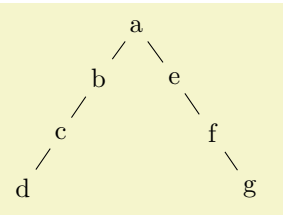
30.2 Specifying Missing Children

In the present section we discuss keys for specifying missing children in a tree. For many certain kind of trees, in particular for binary trees, there are not just "a certain number of children" at each node, but, rather, there is a designated "first" (or "left") child and a "second" (or "right") child. Even if one of these children is missing and a node actually has only one child, the single child will still be a "first" or "second" child and this information should be taken into consideration when drawing a tree.

The first useful key for specifying missing children is `missing number of children` which allows you to state how many children there are, at minimum.

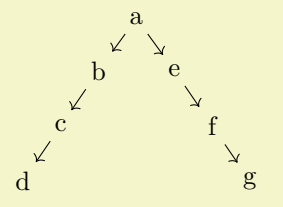
Once the minimum number of children has been set, we still need a way of specifying "missing first children" or, more generally, missing children that are not "at the end" of the list of children. For this, there are three methods:

1. When you use the `child` syntax, you can use the `missing` key with the `child` command to indicate a missing child:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [binary tree layout, level distance=5mm]
\node {a}
child { node {b}
  child { node {c}
    child { node {d} }
} }
child { node {e}
  child [missing]
  child { node {f}
    child [missing]
    child { node {g}
} } };
```

2. When using the `graph` syntax, you can use an "empty node", which really must be completely empty and may not even contain a slash, to indicate a missing node:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz [binary tree layout, level distance=5mm]
\graph { a -> { b -> c -> d, e -> { , f -> { , g} } } };
```

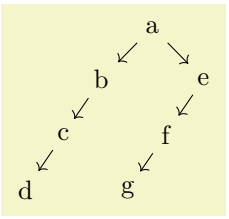
3. You can simply specify the index of a child directly using the key `desired child index`.

`/graph drawing/minimum number of children=`⟨*number*⟩ (initially `0`)

Specifies how many children a tree node must have at least. If there are less, "virtual" children are added. When this key is set to `2` or more, the following happens: We first compute a spanning tree for the graph, see Section 30.3. Then, whenever a node is not a leaf in this spanning tree (when it has at least one child), we add "virtual" or "dummy" nodes as children of the node until the total number of real and dummy children is at least ⟨*number*⟩. If there where at least ⟨*number*⟩ children at the beginning, nothing happens.

The new children are added after the existing children. This means that, for instance, in a tree with ⟨*number*⟩ set to `2`, for every node with a single child, this child will be the first child and the second child will be missing.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout,level distance=5mm]
{ a -> { b->c->d, e->f->g } };
```

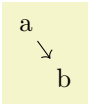
`/graph drawing/desired child index=`⟨*number*⟩

Pass this key to a node to tell the graph drawing engine which child number you "desired" for the node. Whenever all desires for the children of a node are conflict-free, they will all be met; children for which no desired indices were given will remain at their position, whenever possible, but will "make way" for children with a desired position. In detail, the following happens: We first determine the total number of children (real or dummy) needed, which is the maximum of the actual number of children, of the `minimum number of children`, and of the highest desired child index. Then we go over all children that have a desired child index and put they at this position. If the position is already taken (because some other child had the same desired index), the next free position is used with a wrap-around occurring at the end. Next, all children without a desired index are place using the same mechanism, but they want to be placed at the position they had in the original spanning tree.

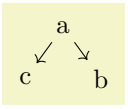
While all of this might sound a bit complicated, the application of the key in a binary tree is pretty straightforward: To indicate that a node is a "right" child in a tree, just add `desired child index=2` to it. This will make it a second child, possibly causing the first child to be missing. If there are two nodes specified as children of a node, by saying `desired child index=`⟨*number*⟩ for one of them, you will cause it be first or second child, depending on ⟨*number*⟩, and cause the *other* child to become the other child.

Since `desired child index=2` is a bit long, the following shortcuts are available: `first`, `second`, `third`, and `fourth`. You might wonder why `second` is used rather than `right`. The reason is that trees may also grow left and right and, additionally, the `right` and `left` keys are already in use for anchoring. Naturally, you can locally redefine them, if you want.

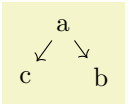
Examples



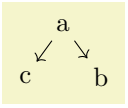

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm]
{ a -> b[second] };
```



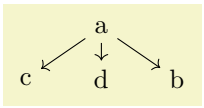

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm]
{ a -> { b[second], c} };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm]
{ a -> { b, c[first]} };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm]
{ a -> { b[second], c[second]} };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [binary tree layout, level distance=5mm]
{ a -> { b[third], c[first], d} };
```

/graph drawing/first=⟨*string*⟩

A shorthand for setting the desired child number to 1.

/graph drawing/second=⟨*string*⟩

A shorthand for setting the desired child number to 2.

/graph drawing/third=⟨*string*⟩

A shorthand for setting the desired child number to 3.

/graph drawing/fourth=⟨*string*⟩

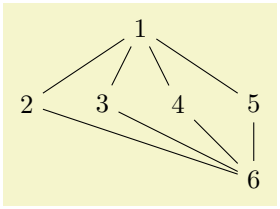
A shorthand for setting the desired child number to 4.

30.3 Spanning Tree Computation

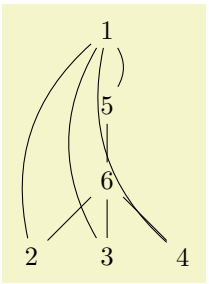
Although the algorithms of this library are tailored to layout trees, they will work for any graph as input. First, if the graph is not connected, it is decomposed into connected components and these are laid out individually. Second, for each component, a spanning tree of the graph is computed first and the layout is computed for this spanning tree; all other edges will still be drawn, but they have no impact on the placement of the nodes. If the graph is already a tree, the spanning tree will be the original graph.

The computation of the spanning tree is a non-trivial process since a non-tree graph has many different possible spanning trees. You can choose between different methods for deciding on a spanning tree, it is even possible to implement new algorithms. (In the future, the computation of spanning trees and the cycle removal in layered graph drawing algorithms will be unified, but, currently, they are implemented differently.)

Selects the (sub)algorithm that is to be used for computing spanning trees whenever this is requested by a tree layout algorithm. The default algorithm is `breadth first spanning tree`.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, breadth first spanning tree]
{
  1 -- {2,3,4,5} -- 6;
};
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, depth first spanning tree]
{
  1 --[bend right] {2,3,4,5 [>bend left]} -- 6;
};
```

/graph drawing/breadth first spanning tree=⟨*string*⟩

This key selects "breadth first" as the (sub)algorithm for computing spanning trees. Note that this key does not cause a graph drawing scope to start; the key only has an effect in conjunction with keys like `tree layout`. The algorithm will be called whenever a graph drawing algorithm needs a spanning tree on which to operate. It works as follows:

1. It looks for a node for which the `root` parameter is set. If there are several such nodes, the first one is used. If there are no such nodes, the first node is used.

   Let call the node determined in this way the *root node*.

2. For every edge, a *priority* is determined, which is a number between 1 and 10. How this happens, exactly, will be explained in a moment. Priority 1 means "most important" while priority 10 means "least important".

3. Starting from the root node, we now perform a breadth first search through the tree, thereby implicitly building a spanning tree: Suppose for a moment that all edges have priority 1. Then, the algorithm works just the way that a normal breadth first search is performed: We keep a queue of to-be-visited nodes and while this queue is not empty, we remove its first node. If this node has not yet been visited, we add all its neighbors at the end of the queue. When a node is taken out of the queue, we make it the child of the node whose neighbor it was when it was added. Since the queue follows the "first in, first out" principle (it is a fifo queue), the children of the root will be all nodes at distance 1 form the root, their children will be all nodes at distance 2, and so on.
4. Now suppose that some edges have a priority different from 1, in which case things get more complicated. We now keep track of one fifo queue for each of the ten possible priorities. When we consider the neighbors of a node, we actually consider all its incident edges. Each of these edges has a certain priority and the neighbor is put into the queue of the edge's priority. Now, we still remove nodes normally from the queue for priority 1; only if this queue is empty and there is still a node in the queue for priority 2 we remove the first element from this queue (and proceed as before). If the second queue is also empty, we try the third, and so on up to the tenth queue. If all queues are empty, the algorithm stops.

The effect of the ten queues is the following: If the edges of priority 1 span the whole graph, a spanning tree consisting solely of these edges will be computed. However, if they do not, once we have visited reachable using only priority 1 edges, we will extend the spanning tree using a priority 2 edge; but then we once switch back to using only priority 1 edges. If neither priority 1 nor priority 2 edges suffice to cover the whole graph, priority 3 edges are used, and so on.

`/graph drawing/depth first spanning tree=`⟨*string*⟩

Works exactly like `breadth first spanning tree` (same handling of priorities), only the queues are now lifo instead of fifo.

`/graph drawing/root=`⟨*boolean*⟩ (default `true`)

This Boolean parameter is used in the computation of spanning trees. When can be set for a node, this node will be used as the root for the spanning tree computation. If several nodes have this option set, the first node will be used.

`/graph drawing/span priority=`⟨*number*⟩

Explicitly sets the "span priority" of an edge to ⟨*number*⟩, which must be a number between `1` and `10`. The priority of edges is used by spanning tree computations, see `breadth first spanning tree`.

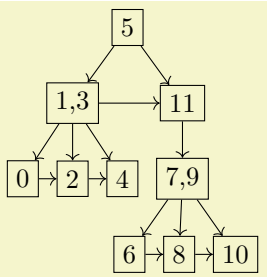
`/graph drawing/span edge=`⟨*string*⟩

An easy-to-remember shorthand for `span priority=1`. When this key is used with an edge, it will always be preferred over other edges

`/graph drawing/no span edge=`⟨*string*⟩

An easy-to-remember shorthand for `span priority=10`. This causes the edge to be used only as a last resort as part of a spanning tree. In the example, we add lots of edges that would normally be preferred in the computation of the spanning tree, but use `no span edge` to cause the algorithm to ignore these edges.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw}, sibling distance=0pt,
              every group/.style={
                default edge kind=->, no span edge,
                path=source}]
{
  5 -> {
    "1,3" -> {0,2,4},
    11    -> {
      "7,9" -> { 6, 8, 10 }
    }
  }
};
```

/graph drawing/span priority ->=⟨*number*⟩ (initially 3)

This key stores the span priority of all edges whose direction is `->`. There are similar keys for all other directions, such as `span priority <-` and so on. When you write

```
graph { a -> b -- c <- [span priority=2] d }
```

the priority of the edge from `a` to `b` would be the current value of the key `span priority ->`, the priority of the edge from `b` to `c` would be the current value of `span priority --`, and the priority of the edge from `c` to `d` would be 2, regardless of the value of `span priority <-`.

The defaults for the priorities are:

- `span priority ->  = 3`
- `span priority --  = 5`
- `span priority <-> = 5`
- `span priority <-  = 8`
- `span priority -!- = 10`

/graph drawing/span priority reversed ->=⟨*number*⟩ (initially 9)

When you write

```
graph { a -> b -- c <- [span priority=2] d }
```

there are, in addition to the priorities indicated above, also further edge priorities: The priority of the (reversed) edge `b` to `a` is `span priority reversed ->`, the priority of the (reversed) edge `c` to `b` is `span priority reversed --`, and the span priority of the reversed edge `d` to `c` is 2, regardless of the value of `span priority reversed <-`.

The defaults for the priorities are:

- `span priority reversed ->  = 9`
- `span priority reversed --  = 5`
- `span priority reversed <-> = 5`
- `span priority reversed <-  = 7`
- `span priority reversed -!- = 10`

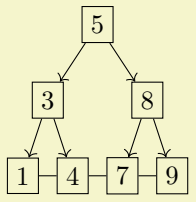
The default priorities are set in such a way, that non-reversed `->` edges have top priorities, `--` and `<->` edges have the same priorities in either direction, and `<-` edges have low priority in either direction (but going `a <- b` from `b` to `a` is given higher priority than going from `a` to `b` via this edge and also higher priority than going from `b` to `a` in `a -> b`).

Keys like `span using directed` change the priorities "en bloc".

/graph drawing/span using directed=⟨*string*⟩

This style sets a priority of 3 for all edges that are directed and "go along the arrow direction", that is, we go from `a` to `b` with a priority of 3 for the cases `a -> b`, `b <- a`, `a <-> b`, and `b <-> a`. This strategy is nice with trees specified with both forward and backward edges.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {trees}
\tikz \graph [tree layout, nodes={draw}, sibling distance=0pt,
              span using directed]
{
  3 <- 5[root] -> 8,
  1 <- 3 -> 4,
  7 <- 8 -> 9,
  1 -- 4 -- 7 -- 9
};
```

/graph drawing/span using all=⟨*string*⟩

Assigns a uniform priority of 5 to all edges.

31 Graph Drawing Algorithms: Layered Layouts

by Till Tantau and Jannis Pohlmann

Graph Drawing Library `layered`

```
\usegdlibrary{layered} % LaTeX and plain TeX
\usegdlibrary[layered] % ConTeXt
```

A "layered" layout of a graph tries to arrange the nodes in consecutive horizontal layers (naturally, by rotating the graph, this can be changed in to vertical layers) such that edges tend to be only between nodes on adjacent layers. Trees, for instance, can always be laid out in this way. This method of laying out a graph is especially useful for hierarchical graphs.

The method implemented in this library is often called the *Sugiyama method*, which is a rather advanced method of assigning nodes to layers and positions on these layers. The same method is also used in the popular GraphViz program, indeed, the implementation in Ti*k*Z is based on the same pseudo-code from the same paper as the implementation used in GraphViz and both programs will often generate the same layout (but not always, as explained below). The current implementation is due to Jannis Pohlmann, who implemented it as part of his Diploma thesis. Please consult this thesis for a detailed explanation of the Sugiyama method and its history:

- Jannis Pohlmann, *Configurable Graph Drawing Algorithms for the Ti*k*Z Graphics Description Language,* Diploma Thesis, Institute of Theoretical Computer Science, Universität zu Lübeck, 2011.

  Available online via `http://www.tcs.uni-luebeck.de/downloads/papers/2011/2011-configurable-graph-drawing-algorithms-jannis-pohlmann.pdf`

  (Note that since the publication of this thesis some option names have been changed. Most noticeably, the option name `layered drawing` was changed to `layered layout`, which is somewhat more consistent with other names used in the graph drawing libraries. Furthermore, the keys for choosing individual algorithms for the different algorithm phases, have all changed.)

The Sugiyama methods lays out a graph in five steps:

1. Cycle removal.
2. Layer assignment (sometimes called node ranking).
3. Crossing minimization (also referred to as node ordering).
4. Node positioning (or coordinate assignment).
5. Edge routing.

It turns out that behind each of these steps there lurks an NP-complete problem, which means, in practice, that each step is impossible to perform optimally for larger graphs. For this reason, heuristics and approximation algorithms are used to find a "good" way of performing the steps.

A distinctive feature of Pohlmann's implementation of the Sugiyama method for Ti*k*Z is that the algorithms used for each of the steps can easily be exchanged, just specify a different option. For the user, this means that by specifying a different option and thereby using a different heuristic for one of the steps, a better layout can often be found. For the researcher, this means that one can very easily test new approaches and new heuristics without having to implement all of the other steps anew.

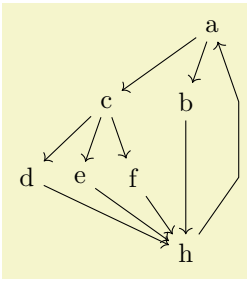
31.1 The Modular Sugiyama Method

`/graph drawing/layered layout=`⟨*string*⟩

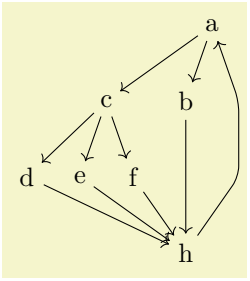
The `layered layout` is the key used to select the modular Sugiyama layout algorithm. This algorithm consists of five consecutive steps, each of which can be configured independently of the other ones (how this is done is explained later in this section). Naturally, the "best" heuristics are selected by default, so there is typically no need to change the settings, but what is the "best" method for one graph need not be the best one for another graph.

As can be seen in the first example, the algorithm will not only position the nodes of a graph, but will also perform an edge routing. This will look visually quite pleasing if you add the `rounded corners` option:

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout, sibling distance=7mm]
{
  a -> {
    b,
    c -> { d, e, f }
  } ->
  h ->
  a
};
```

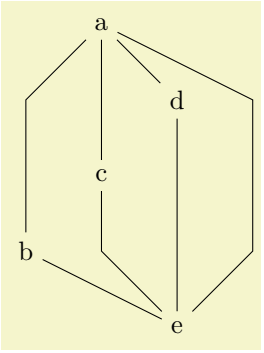



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz [rounded corners] \graph [layered layout, sibling distance=7mm]
{
  a -> {
    b,
    c -> { d, e, f }
  } ->
  h ->
  a
};
```

`/graph drawing/minimum layers=`⟨*number*⟩ (initially `1`)

The minimum number of levels that an edge must span. It is a bit of the opposite of the `weight` parameter: While a large `weight` causes an edge to become shorter, a larger `minimum layers` value causes an edge to be longer.

Example

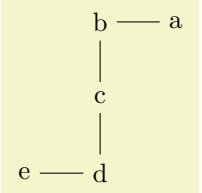



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout] {
  a -- {b [> minimum layers=3], c, d} -- e -- a;
};
```

`/graph drawing/same layer` (style)

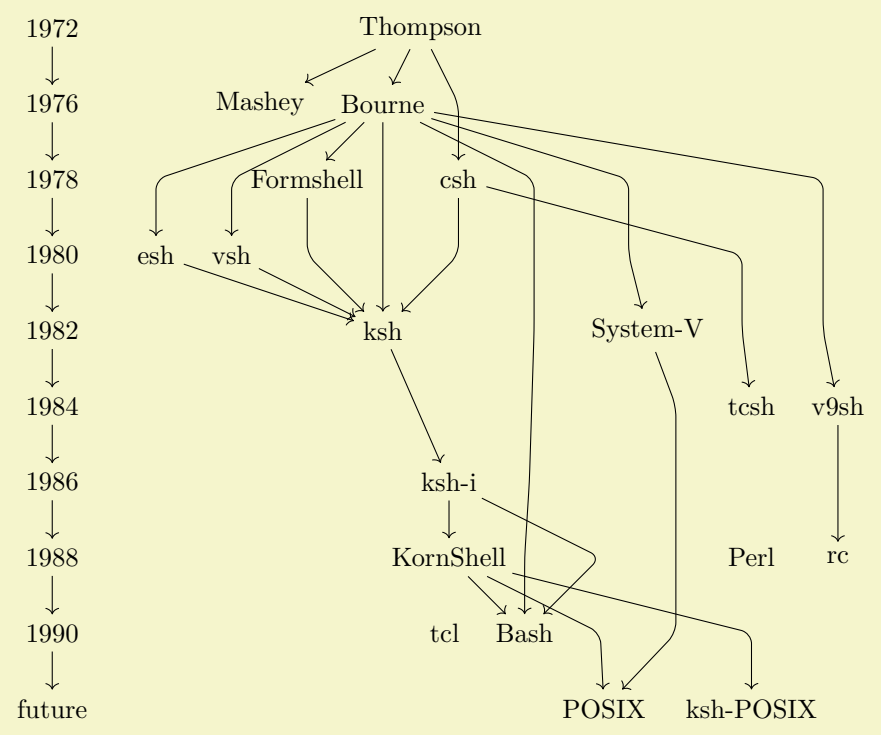
The `same layer` collection allows you to enforce that several nodes a on the same layer of a layered layout (this option is also known as `same rank`). You use it like this:

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz \graph [layered layout] {
  a -- b -- c -- d -- e;

  { [same layer] a, b };
  { [same layer] d, e };
};
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {layered}
\tikz [rounded corners] \graph [layered layout] {
  1972 -> 1976 -> 1978 -> 1980 -> 1982 -> 1984 -> 1986 -> 1988 -> 1990 -> future;

  { [same layer] 1972, Thompson };
  { [same layer] 1976, Mashey, Bourne },
  { [same layer] 1978, Formshell, csh },
  { [same layer] 1980, esh, vsh },
  { [same layer] 1982, ksh, "System-V" },
  { [same layer] 1984, v9sh, tcsh },
  { [same layer] 1986, "ksh-i" },
  { [same layer] 1988, KornShell ,Perl, rc },
  { [same layer] 1990, tcl, Bash },
  { [same layer] "future", POSIX, "ksh-POSIX" },

  Thompson -> { Mashey, Bourne, csh -> tcsh},
  Bourne -> { ksh, esh, vsh, "System-V", v9sh -> rc, Bash},
  { "ksh-i", KornShell } -> Bash,
  { esh, vsh, Formshell, csh } -> ksh,
  { KornShell, "System-V" } -> POSIX,
  ksh -> "ksh-i" -> KornShell -> "ksh-POSIX",
  Bourne -> Formshell,

  { [edge={draw=none}]
    Bash -> tcl,
    KornShell -> Perl
  }
};
```

31.2 Cycle Removal

The Sugiyama method works only on directed *acyclic* graphs. For this reason, if the input graph is not (yet) acyclic, a number of edges need to be redirected so that acyclicity arises. In the following, the different options that allow you to fine-tune this process are documented.

`/graph drawing/depth first cycle removal=`⟨*string*⟩

Selects a cycle removal algorithm that is especially appropriate for graphs specified "by hand". When graphs are created by humans manually, one can make assumptions about the input graph that would otherwise not be possible. For instance, it seems reasonable to assume that the order in which nodes

and edges are entered by the user somehow reflects the natural flow the user has had in mind for the graph.

In order to preserve the natural flow of the input graph, Gansner et al. propose to remove cycles by performing a series of depth-first searches starting at individual nodes in the order they appear in the graph. This algorithm implicitly constructs a spanning tree of the nodes reached during the searches. It thereby partitions the edges of the graph into tree edges and non-tree edges. The non-tree edges are further subdivided into forward edges, cross edges, and back edges. Forward edges point from a tree nodes to one of their descendants. Cross edges connect unrelated branches in the search tree. Back edges connect descendants to one of their ancestors. It is not hard to see that reversing back edges will not only introduce no new cycles but will also make any directed graph acyclic. Gansner et al. argue that this approach is more stable than others in that fewer inappropriate edges are reversed compared to other methods, despite the lack of a provable upper bound for the number of reversed edges.

See section 4.1.1 of Pohlmann's Diplom thesis for more details.

This is the default algorithm for cycle removals.

`/graph drawing/prioritized greedy cycle removal=`⟨*string*⟩

This algorithm implements a greedy heuristic of Eades et al. for cycle removal that prioritizes sources and sinks. See section 4.1.1 of Pohlmann's Diploma theses for details.

`/graph drawing/greedy cycle removal=`⟨*string*⟩

This algorithm implements a greedy heuristic of Eades et al. for cycle removal that prioritizes sources and sinks. See section 4.1.1 of Pohlmann's Diploma theses for details.

`/graph drawing/naive greedy cycle removal=`⟨*string*⟩

This algorithm implements a greedy heuristic of Berger and Shor for cycle removal. It is not really compared to the other heuristics and only included for demonstration purposes. See section 4.1.1 of Pohlmann's Diploma theses for details.

`/graph drawing/random greedy cycle removal=`⟨*string*⟩

This algorithm implements a randomized greedy heuristic of Berger and Shor for cycle removal. It, too, is not really compared to the other heuristics and only included for demonstration purposes. See section 4.1.1 of Pohlmann's Diploma theses for details.

31.3 Layer Assignment (Node Ranking)

Algorithms for producing layered drawings place nodes on discrete layers from top to bottom. Layer assignment is the problem of finding a partition so that for all edges $e = (u, v) \in E(G)$ the equation $layer(u) < layer(v)$ holds. Such a partition is called a *layering*. This definition can be extended by introducing edge weights or priorities and minimum length constraints which has practical applications and allows users to fine-tune the results.

For more details, please see Section 4.1.2 of Pohlmann's Diploma thesis.

`/graph drawing/linear optimization layer assignment=`⟨*string*⟩

This layer assignment method, due to Gasner et al., is based on a linear optimization problem. For more details, please see Section 4.1.2 of Pohlmann's Diploma thesis.

This is the default algorithm for layer assignments.

`/graph drawing/minimum height layer assignment=`⟨*string*⟩

This layer assignment method minimizes the height of the resulting graph. For more details, please see Section 4.1.3 of Pohlmann's Diploma thesis.

31.4 Crossing Minimization (Node Ordering)

The number of edge crossings in a layered drawing is determined by the ordering of nodes at each of its layers. Therefore, crossing minimization is the problem of reordering the nodes at each layer so that the overall number of edge crossings is minimized. The crossing minimization step takes a proper layering where every edge connects nodes in neighbored layers, allowing algorithms to minimize crossings layer by layer rather

than all at once. While this does not reduce the complexity of the problem, it does make it considerably easier to understand and implement. Techniques based on such an iterative approach are also known as layer-by-layer sweep methods. They are used in many popular heuristics due to their simplicity and the good results they produce.

Sweeping refers to moving up and down from one layer to the next, reducing crossings along the way. In layer-by-layer sweep methods, an initial node ordering for one of the layers is computed first. Depending on the sweep direction this can either be the first layer or the last; in rare occasions the layer in the middle is used instead. Followed by this, the actual layer-by-layer sweep is performed. Given an initial ordering for the first layer L_1, a downward sweep first holds the nodes in L_1 fixed while reordering the nodes in the second layer L_2 to reduce the number of crossings between L_1 and L_2. It then goes on to reorder the third layer while holding the second layer fixed. This is continued until all layers except for the first one have been examined. Upward sweeping and sweeping from the middle work analogous.

Obviously, the central aspect of the layer-by-layer sweep is how the nodes of a specific layer are reordered using a neighbored layer as a fixed reference. This problem is known as one-sided crossing minimization, which unfortunately is NP-hard. In the following various heuristics to solve this problem are presented.

For more details, please see Section 4.1.4 of Pohlmann's Diploma thesis.

`/graph drawing/sweep crossing minimization=`⟨*string*⟩

Gansner et al. combine an initial ordering based on a depth-first search with the median and greedy switch heuristics applied in the form of an alternating layer-by-layer sweep based on a weighted median. For more details, please see Section 4.1.4 of Pohlmann's Diploma thesis.

This is the default algorithm for crossing minimization.

31.5 Node Positioning (Coordinate Assignment)

The second last step of the Sugiyama method decides about the final x- and y-coordinates of the nodes. The main objectives of this step are to position nodes so that the number of edge bends is kept small and edges are drawn as vertically as possible. Another goal is to avoid node and edge overlaps which is crucial in particular if the nodes are allowed to have non-uniform sizes. The y-coordinates of the nodes have no influence on the number of bends. Obviously, nodes need to be separated enough geometrically so that they do not overlap. It feels natural to aim at separating all layers in the drawing by the same amount. Large nodes, however, may force node positioning algorithms to override this uniform level distance in order to avoid overlaps.

For more details, please see Section 4.1.2 of Pohlmann's Diploma thesis.

`/graph drawing/linear optimization node positioning=`⟨*string*⟩

This node positioning method, due to Gasner et al., is based on a linear optimization problem. For more details, please see Section 4.1.3 of Pohlmann's Diploma thesis.

This is the default algorithm for layer assignments.

31.6 Edge Routing

The original layered drawing method described by Eades and Sugiyama in does not include the routing or shaping of edges as a main step. This makes sense if all nodes have the same size and shape. In practical scenarios, however, this assumption often does not hold. In these cases, advanced techniques may have to be applied in order to avoid overlaps of nodes and edges.

For more details, please see Section 4.1.5 of Pohlmann's Diploma thesis.

`/graph drawing/polyline layer edge routing=`⟨*string*⟩

This edge routing algorithm uses polygonal lines to connect nodes. For more details, please see Section 4.1.5 of Pohlmann's Diploma thesis.

This is the default algorithm for edge routing.

32 Graph Drawing Algorithms: Force-Based Methods

by Till Tantau and Jannis Pohlmann

Graph Drawing Library `force`

```
\usegdlibrary{force} % LaTeX and plain TeX
\usegdlibrary[force] % ConTeXt
```

Nature creates beautiful graph layouts all the time. Consider a spider's web: Nodes are connected by edges in a visually most pleasing manner (if you ignore the spider in the middle). The layout of a spider's web is created just by the physical forces exerted by the threads. The idea behind force-based graph drawing algorithms is to mimic nature: We treat edges as threads that exert forces and simulate into which configuration the whole graph is "pulled" by these forces.

When you start thinking about for a moment, it turns out that there are endless variations of the force model. All of these models have the following in common, however:

- "Forces" pull and push at the nodes in different directions.
- The effect of these forces is simulated by iteratively moving all the nodes simultaneously a little in the direction of the forces and by then recalculating the forces.
- The iteration is stopped either after a certain number of iterations or when a *global energy minimum* is reached (a very scientific way of saying that nothing happens anymore).

The main difference between the different force-based approaches is how the forces are determined. Here are some ideas what could cause a force to be exerted between two nodes (and there are more):

- If the nodes are connected by an edge, one can treat the edge as a "spring" that has a "natural spring dimension". If the nodes are nearer than the spring dimension, they are push apart; if they are farther away than the spring dimension, they are pulled together.
- If two nodes are connected by a path of a certain length, the nodes may "wish to be at a distance proportional to the path length". If they are nearer, they are pushed apart; if they are farther, they are pulled together. (This is obviously a generalization of the previous idea.)
- There may be a general force field that pushes nodes apart (an electrical field), so that nodes do not tend to "cluster".
- There may be a general force field that pulls nodes together (a gravitational field), so that nodes are not too loosely scattered.
- There may be highly nonlinear forces depending on the distance of nodes, so that nodes very near to each get pushed apart strongly, but the effect wears of rapidly at a distance. (Such forces are known as strong nuclear forces.)
- There rotational forces caused by the angles between the edges leaving a node. Such forces try to create a *perfect angular resolution* (a very scientific way of saying that all angles at a node are equal).

Force-based algorithms combine one or more of the above ideas into a single algorithm that uses "good" formulas for computing the forces.

Currently, three algorithms are implemented in this library, two of which are from the first of the following paper, while the third is from the third paper:

- Y. Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 2006.
- C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Graph Drawing*, Lecture Notes in Computer Science, 1984:31–55, 2001.

Our implementation is described in detail in the following diploma thesis:

- Jannis Pohlmann, *Configurable Graph Drawing Algorithms for the TikZ Graphics Description Language,* Diploma Thesis, Institute of Theoretical Computer Science, Universität zu Lübeck, 2011.

  Online at `http://www.tcs.uni-luebeck.de/downloads/papers/2011/2011-configurable-graph-drawing-algorithms-jannis-pohlmann.pdf`

In the future, I hope that most, if not all, of the force-based algorithms become "just configuration options" of a general force-based algorithm similar to the way the modular Sugiyama method is implemented in the `layered` graph drawing library.

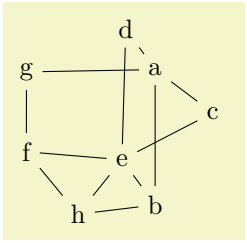
32.1 Controlling and Configuring Force-Based Algorithms

All force-based algorithms are based on a general pattern which we detail in the following. Numerous options can be used to influence the behavior of this general pattern; more specific options that apply only to individual algorithms are explained along with these algorithms.

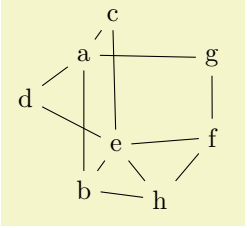
The vertices are initially laid out in a random configuration. Then the configuration is annealed to find a configuration of minimal energy. To avoid getting stuck in a local minimum or at a saddle point, random forces are added. All of this makes the final layout extremely susceptible to changes in the random numbers. To achieve a certain stability of the results, you should fix the random seed. However, in the recent past Lua has switched its random number generator, which means that you won't get the same sequence of random numbers as in a previous version, even for identical seed. If you rely on the long-term stability of vertex placement, you should consider using a different layout. With the spring layout you have to assume that the layout will be random.

32.1.1 Start Configuration

Currently, the start configuration for force-based algorithms is a random distribution of the vertices. You can influence it by changing the `random seed`:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [random seed=10, spring layout] {
  a -- {b, c, d} -- e -- f -- {g,h} -- {a,b,e};
};
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [random seed=11, spring layout] {
  a -- {b, c, d} -- e -- f -- {g,h} -- {a,b,e};
};
```

Other methods, like a planar preembedding, are not implemented currently.

32.1.2 The Iterative Process and Cooling

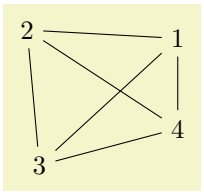
`/graph drawing/iterations=`⟨*number*⟩ (initially `500`)

Limits the number of iterations of algorithms for force-based layouts to ⟨*number*⟩. Depending on the characteristics of the input graph and the parameters chosen for the algorithm, minimizing the system energy may require many iterations.

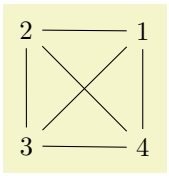
In these situations it may come in handy to limit the number of iterations. This feature can also be useful to draw the same graph after different iterations and thereby demonstrate how the spring or spring-electrical algorithm improves the drawing step by step.

The examples shows two drawings generated using two different `iteration` limits.

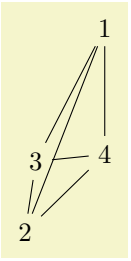
Examples



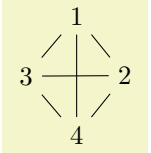

```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, iterations=10]  { subgraph K_n [n=4] };
```




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, iterations=500] { subgraph K_n [n=4] };
```




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, iterations=10]
  { subgraph K_n [n=4] };
```




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, iterations=500]
  { subgraph K_n [n=4] };
```

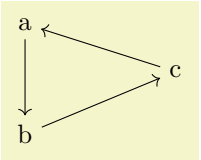
`/graph drawing/initial step length=`⟨*length*⟩ (initially `0`)

This parameter specifies the amount by which nodes will be displaced in each iteration, initially. If set to `0` (which is the default), an appropriate value is computed automatically.

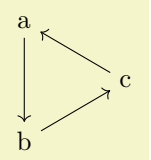
`/graph drawing/cooling factor=`⟨*number*⟩ (initially `0.95`)

This parameter helps in controlling how layouts evolve over time. It is used to gradually reduce the step size between one iteration to the next. A small positive cooling factor ≥ 0 means that the movement of nodes is quickly or abruptly reduced, while a large cooling factor ≤ 1 allows for a smoother step by step layout refinement at the cost of more iterations. The following example demonstrates how a smaller cooling factor may result in a less balanced drawing. By default, Hu2006 spring, Hu2006 spring electrical, and Walshaw2000 spring electrical use a cooling factor of `0.95`.

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, cooling factor=0.1]
  { a -> b -> c -> a };
```

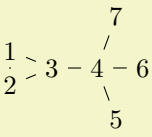



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, cooling factor=0.5]
  { a -> b -> c -> a };
```

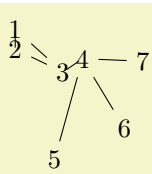
`/graph drawing/convergence tolerance=`⟨*number*⟩ (initially `0.01`)

All spring and spring-electrical algorithms implemented in the thesis terminate as soon as the maximum movement of any node drops below $k \cdot \langle tolerance \rangle$. This tolerance factor can be changed with the convergence tolerance option:

Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, convergence tolerance=0.001]
  { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };
```

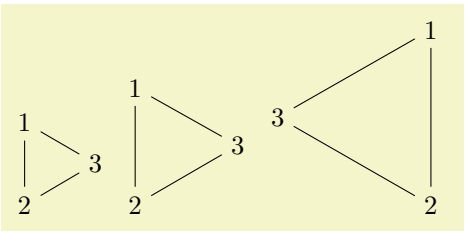



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, convergence tolerance=1.0]
  { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };
```

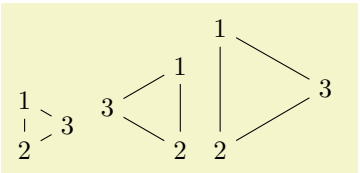
32.1.3 Forces and Their Effects: Springs

The most important parameter of springs is their "natural length", which can be configured using the general-purpose `node distance` parameter. It is the "equilibrium length" of a spring between two nodes in the graph. When an edge has this length, no forces will "push" or "pull" along the edge.

The following examples shows how a simple graph can be scaled by changing the `node distance`:




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, node distance=7mm] { subgraph C_n[n=3] };
\tikz \graph [spring layout]                    { subgraph C_n[n=3] };
\tikz \graph [spring layout, node distance=15mm]{ subgraph C_n[n=3] };
```




```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, node distance=0.7cm] { subgraph C_n[n=3] };
\tikz \graph [spring electrical layout]                      { subgraph C_n[n=3] };
\tikz \graph [spring electrical layout, node distance=1.5cm] { subgraph C_n[n=3] };
```

`/graph drawing/spring constant=`⟨*number*⟩ (initially `0.01`)

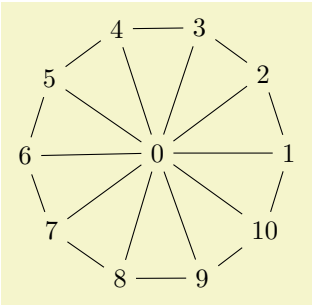
The "spring constant" is a factor from Hooke's law describing the "stiffness" of a spring. This factor is used inside spring-based algorithms to determine how strongly edges "pull" and "push" at the nodes they connect.

32.1.4 Forces and Their Effects: Electrical Repulsion

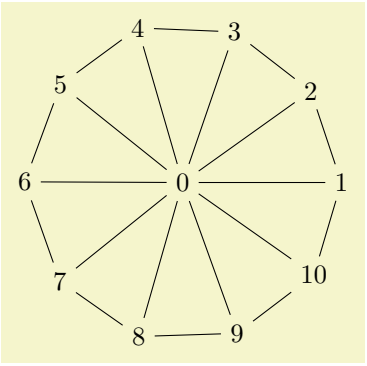
`/graph drawing/electric charge=`⟨*number*⟩ (initially `1`)

Defines the electric charge of the node. The stronger the `electric charge` of a node the stronger the repulsion between the node and others in the graph. A negative `electric charge` means that other nodes are further attracted to the node rather than repulsed, although in theory this effect strongly depends on how the `spring electrical layout` algorithm works. Two typical effects of increasing the `electric charge` are distortion of symmetries and an upscaling of the drawings.

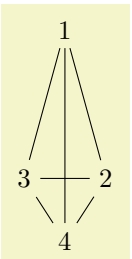
Examples




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, horizontal=0 to 1]
  { 0 [electric charge=1] -- subgraph C_n [n=10] };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, horizontal=0 to 1]
  { 0 [electric charge=5] -- subgraph C_n [n=10] };
```




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout, horizontal=0 to 1]
  { [clique] 1 [electric charge=5], 2, 3, 4 };
```

`/graph drawing/electric force order=`⟨*number*⟩ (initially `1`)

Sometimes, when drawing symmetric and mesh-like graphs, the peripheral distortion caused by long-range electric forces may be undesired. Some electric force models allow to reduce long-range forces and distortion effects by increasing the order (exponent) of electric forces. Values between 0 and 1 increase long-range electric forces and the scaling of the generated layouts. Value greater than 1 decrease long-range electric forces and results in shrinking drawings.

`/graph drawing/approximate remote forces=`⟨*boolean*⟩ (default `true`)

Force based algorithms often need to compute a force for each pair of vertices, which, for larger numbers of vertices, can lead to a significant time overhead. This problem can be addressed by approximating

these forces: For a vertex far removed from a cluster of vertices, instead of computing the force contribution of each vertex of the cluster individually, we form a sort of "supervertex" at the "gravitational center" of the cluster and then compute only the force between this supervertex and the single vertex.

Remark: Currently, the implementation seems to be broken, at least the results are somewhat strange when this key is used.

32.1.5 Coarsening

`/graph drawing/coarsen=`⟨*boolean*⟩ (default `true`, initially `true`)

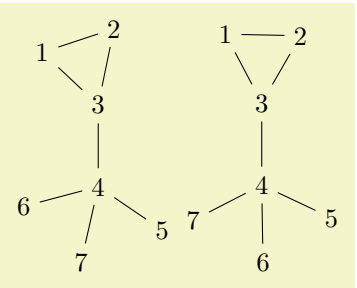
Defines whether or not a multilevel approach is used that iteratively coarsens the input graph into graphs $G_1, \ldots, G_l$ with a smaller and smaller number of nodes. The coarsening stops as soon as a minimum number of nodes is reached, as set via the `minimum coarsening size` option, or if, in the last iteration, the number of nodes was not reduced by at least the ratio specified via `downsize ratio`. A random initial layout is computed for the coarsest graph G_l first. Afterwards, it is laid out by computing the attractive and repulsive forces between its nodes.

In the subsequent steps, the previous coarse graph G_{l-1} is restored and its node positions are interpolated from the nodes in G_l. The graph G_{l-1} is again laid out by computing the forces between its nodes. These steps are repeated with $G_{l-2}, \ldots, G_1$ until the original input graph G_0 has been restored, interpolated and laid out.

The idea behind this approach is that, by arranging recursively formed supernodes first and then interpolating and arranging their subnodes step by step, the algorithm is less likely to settle in a local energy minimum (of which there can be many, particularly for large graphs). The quality of the drawings with coarsening enabled is expected to be higher than graphics where this feature is not applied.

The following example demonstrates how coarsening can improve the quality of graph drawings generated with Walshaw's algorithm `spring electrical layout'`.

Example



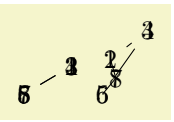

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring electrical layout', coarsen=false, vertical=3 to 4]
  {
    { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 }
  };

\tikz \graph [spring electrical layout', coarsen, vertical=3 to 4]
  {
    { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 }
  };
```

`/graph drawing/minimum coarsening size=`⟨*number*⟩ (initially 2)

Defines the minimum number of nodes down to which the graph is coarsened iteratively. The first graph that has a smaller or equal number of nodes becomes the coarsest graph G_l, where l is the number of coarsening steps. The algorithm proceeds with the steps described in the documentation of the `coarsen` option. In the following example the same graph is coarsened down to two and four nodes, respectively. The layout of the original graph is interpolated from the random initial layout and is not improved further because the forces are not computed (0 iterations). Thus, in the two graphs, the nodes are placed at exactly two and four coordinates in the final drawing.

Example



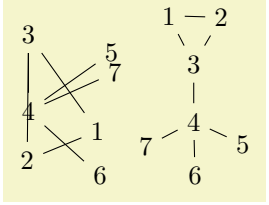
```
\usetikzlibrary {graphs.standard,graphdrawing} \usegdlibrary {force}
\tikz \graph [spring layout, iterations=0,
              minimum coarsening size=2]
  { subgraph C_n [n=8] };

\tikz \graph [spring layout, iterations=0,
              minimum coarsening size=4]
  { subgraph C_n [n=8] };
```

/graph drawing/downsize ratio=⟨*number*⟩ (initially 0.25)

Minimum ratio between 0 and 1 by which the number of nodes between two coarse graphs G_i and G_{i+1} need to be reduced in order for the coarsening to stop and for the algorithm to use G_{i+1} as the coarsest graph G_l. Aside from the input graph, the optimal value of `downsize ratio` mostly depends on the coarsening scheme being used. Possible schemes are `collapse independent edges` and `connect independent nodes`. Increasing this option possibly reduces the number of coarse graphs computed during the coarsening phase as coarsening will stop as soon as a coarse graph does not reduce the number of nodes substantially. This may speed up the algorithm but if the size of the coarsest graph G_l is much larger than `minimum coarsening size`, the multilevel approach may not produce drawings as good as with a lower `downsize ratio`.

Example



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {force}
% 1. ratio too high, coarsening stops early, benefits are lost
\tikz \graph [spring electrical layout',
              downsize ratio=1.0,
              node distance=7mm, vertical=3 to 4]
  { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };

% 2. ratio set to default, coarsening benefits are visible
\tikz \graph [spring electrical layout',
              downsize ratio=0.2,
              node distance=7mm, vertical=3 to 4]
  { { [clique] 1, 2 } -- 3 -- 4 -- { 5, 6, 7 } };
```

32.2 Spring Layouts

/graph drawing/spring layout=⟨*string*⟩

This key selects Hu's 2006 spring layout with appropriate settings for some parameters.

/graph drawing/spring Hu 2006 layout=⟨*string*⟩

Implementation of a spring graph drawing algorithm based on a paper by Hu.

- Y. Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 2006.

There are some modifications compared to the original algorithm, see the Diploma thesis of Pohlmann for details.

32.3 Spring Electrical Layouts

/graph drawing/spring electrical layout=⟨*string*⟩

This key selects Hu's 2006 spring electrical layout with appropriate settings for some parameters.

/graph drawing/spring electrical layout'=⟨*string*⟩

This key selects Walshaw's 2000 spring electrical layout with appropriate settings for some parameters.

/graph drawing/spring electrical Hu 2006 layout=⟨*string*⟩

Implementation of a spring electrical graph drawing algorithm based on a paper by Hu.

- Y. Hu. Efficient, high-quality force-directed graph drawing. *The Mathematica Journal*, 2006.

There are some modifications compared to the original algorithm, see the Diploma thesis of Pohlmann for details.

/graph drawing/spring electrical Walshaw 2000 layout=⟨*string*⟩

Implementation of a spring electrical graph drawing algorithm based on a paper by Walshaw.

- C. Walshaw. A multilevel algorithm for force-directed graph drawing. In J. Marks, editor, *Graph Drawing*, Lecture Notes in Computer Science, 1984:31–55, 2001.

The following modifications compared to the original algorithm were applied:

- An iteration limit was added.
- The natural spring length for all coarse graphs is computed based on the formula presented by Walshaw, so that the natural spring length of the original graph (coarse graph 0) is the same as the value requested by the user.
- Users can define custom node and edge weights.
- Coarsening stops when $|V(G_i + 1)|/|V(G_i)| < p$ where $p = 0.75$.
- Coarsening stops when the maximal matching is empty.
- The runtime of the algorithm is improved by use of a quadtree data structure like Hu does in his algorithm.
- A limiting the number of levels of the quadtree is not implemented.

33 Graph Drawing Algorithms: Circular Layouts

by Till Tantau

Graph Drawing Library `circular`

```
\usegdlibrary{circular} % LaTeX and plain TeX
\usegdlibrary[circular] % ConTeXt
```

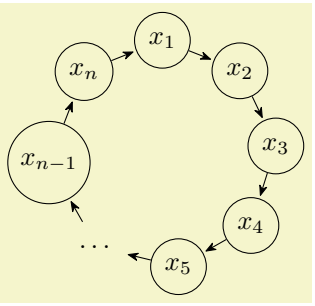
"Circular" graph drawing algorithms arrange the nodes of a graph on one of more circles.

`/graph drawing/simple necklace layout=`⟨*string*⟩

This simple layout arranges the nodes in a circle, which is especially useful for drawing, well, circles of nodes. The name `simple necklace layout` is reminiscent of the more general "necklace layout", a term coined by Speckmann and Verbeek in their paper

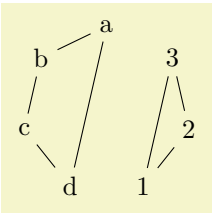
- Bettina Speckmann and Kevin Verbeek, Necklace Maps, *IEEE Transactions on Visualization and Computer Graphics,* 16(6):881–889, 2010.

For a `simple necklace layout`, the centers of the nodes are placed on a counter-clockwise circle, starting with the first node at the `grow` direction (for `grow'`, the circle is clockwise). The order of the nodes is the order in which they appear in the graph, the edges are not taken into consideration, unless the `componentwise` option is given.

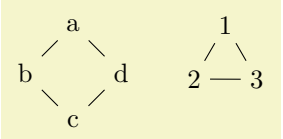



```
\usetikzlibrary {arrows.meta,graphs,graphdrawing} \usegdlibrary {circular}
\tikz[>={Stealth[round,sep]}]
  \graph [simple necklace layout, grow'=down, node sep=1em,
          nodes={draw,circle}, math nodes]
  {
    x_1 -> x_2 -> x_3 -> x_4 ->
    x_5 -> "\dots"[draw=none] -> "x_{n-1}" -> x_n -> x_1
  };
```

When you give the `componentwise` option, the graph will be decomposed into connected components, which are then laid out individually and packed using the usual component packing mechanisms:




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout] {
  a -- b -- c -- d -- a,
  1 -- 2 -- 3 -- 1
};
```



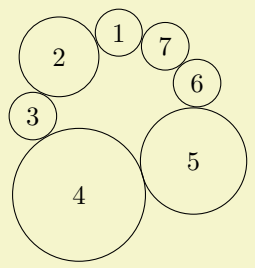

```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {circular}
\tikz \graph [simple necklace layout, componentwise] {
  a -- b -- c -- d -- a,
  1 -- 2 -- 3 -- 1
};
```

The nodes are placed in such a way that

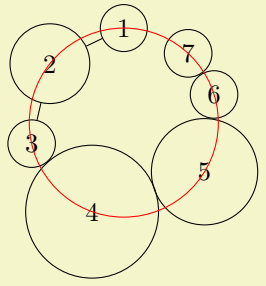
1. The (angular) distance between the centers of consecutive nodes is at least `node distance`,
2. the distance between the borders of consecutive nodes is at least `node sep`, and
3. the radius is at least `radius`.

The radius of the circle is chosen near-minimal such that the above properties are satisfied. To be more precise, if all nodes are circles, the radius is chosen optimally while for, say, rectangular nodes there may be too much space between the nodes in order to satisfy the second condition.

Examples

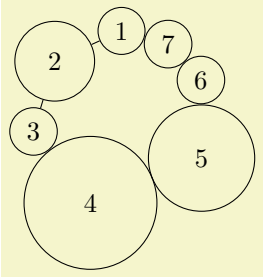



```
\tikz \graph [simple necklace layout,
              node sep=0pt, node distance=0pt,
              nodes={draw,circle}]
{ 1 -- 2 [minimum size=30pt] -- 3 --
  4 [minimum size=50pt] -- 5 [minimum size=40pt] -- 6 -- 7 };
```

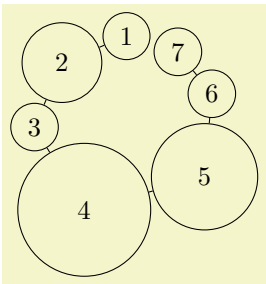



```
\begin{tikzpicture}[radius=1.25cm]
  \graph [simple necklace layout,
          node sep=0pt, node distance=0pt,
          nodes={draw,circle}]
  { 1 -- 2 [minimum size=30pt] -- 3 --
    4 [minimum size=50pt] -- 5 [minimum size=40pt] -- 6 -- 7 };

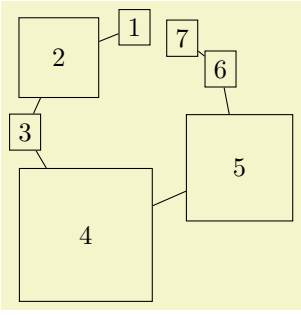
  \draw [red] (0,-1.25) circle [];
\end{tikzpicture}
```




```
\tikz \graph [simple necklace layout,
              node sep=0pt, node distance=1cm,
              nodes={draw,circle}]
{ 1 -- 2 [minimum size=30pt] -- 3 --
  4 [minimum size=50pt] -- 5 [minimum size=40pt] -- 6 -- 7 };
```




```
\tikz \graph [simple necklace layout,
              node sep=2pt, node distance=0pt,
              nodes={draw,circle}]
{ 1 -- 2 [minimum size=30pt] -- 3 --
  4 [minimum size=50pt] -- 5 [minimum size=40pt] -- 6 -- 7 };
```




```
\tikz \graph [simple necklace layout,
              node sep=0pt, node distance=0pt,
              nodes={rectangle,draw}]
{ 1 -- 2 [minimum size=30pt] -- 3 --
  4 [minimum size=50pt] -- 5 [minimum size=40pt] -- 6 -- 7 };
```

34 Graph Drawing Layouts: Phylogenetic Trees

by Sarah Mäusle and Till Tantau

Graph Drawing Library phylogenetics

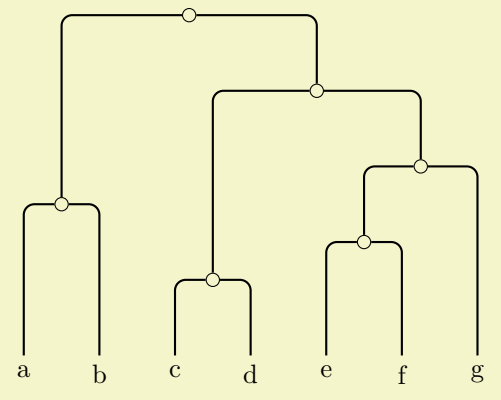
```
\usegdlibrary{phylogenetics} % LaTeX and plain TeX
\usegdlibrary[phylogenetics] % ConTeXt
```

A phylogenetic tree (or network) depicts the evolutionary history of species or, more generally, so called taxa. The present library includes a number of algorithms for drawing phylogenetic trees.

/graph drawing/phylogenetic tree layout=⟨*string*⟩

Layout for drawing phylogenetic trees. ...

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout, upgma,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

34.1 Generating a Phylogenetic Tree

...

When a phylogenetic tree is generated, new nodes and edges get created. In order to give you a chance at styling them, the following styles are executed:

/graph drawing/phylogenetic inner node (style, no value)

The style is added every newly created inner node. In this manual, this key is set to:

```
\pgfgdset{phylogenetic inner node/.style={
    /tikz/.cd, draw, circle, inner sep=0pt, minimum size=5pt
  }
}
```

/graph drawing/phylogenetic edge=⟨*length*⟩ (style, no default)

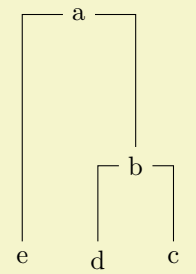
The style is added every newly created phylogenetic edge. The ⟨*length*⟩ will be set to the computed evolutionary length of the edge. In this manual, this key is set to:

```
\pgfgdset{phylogenetic edge/.style={
    /tikz/.cd, thick, rounded corners
  }
}
```

/graph drawing/**phylogenetic tree by author**=⟨*string*⟩

When this key is used, the phylogenetic tree must be specified by the author (rather than being generated algorithmically). A spanning tree of the input graph will be computed first (it must be connected, otherwise errors will result). The evolutionary length of the edges must be specified through the use of the `length` key for each edge.

Example

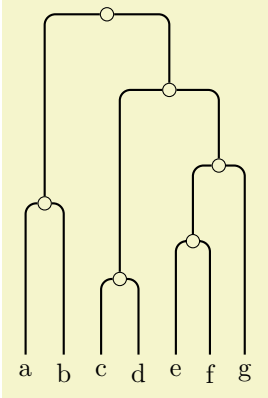



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics,trees}
\tikz \graph [phylogenetic tree layout] {
  a -- {
    b [>length=2] --[length=1] { c, d },
    e [>length=3]
  }
};
```

/graph drawing/**unweighted pair group method using arithmetic averages**=⟨*string*⟩

The UPGMA (Unweighted Pair Group Method using arithmetic Averages) algorithm of Sokal and Michener, 1958. It generates a graph on the basis of such a distance matrix by generating nodes and computing the edge lengths. This algorithm uses a distance matrix, ideally an ultrametric one, to compute the graph.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout, sibling distance=0pt, sibling sep=2pt,
              unweighted pair group method using arithmetic averages,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

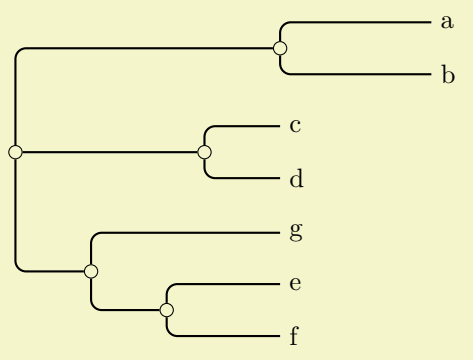
/graph drawing/**upgma**=⟨*string*⟩

An shorthand for `unweighted pair group method using arithmetic averages`

/graph drawing/**balanced minimum evolution**=⟨*string*⟩

The BME (Balanced Minimum Evolution) algorithm tries to minimize the total tree length. This algorithm is from Desper and Gascuel, *Fast and Accurate Phylogeny Reconstruction Algorithms Based on the Minimum-Evolution Principle*, 2002. The tree is built in a way that minimizes the total tree length. The leaves are inserted into the tree one after another, creating new edges and new nodes. After every insertion the distance matrix has to be updated.

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout,
              balanced minimum evolution,
              grow'=right, sibling distance=0pt,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

/graph drawing/balanced nearest neighbour interchange=⟨*string*⟩

The BNNI (Balanced Nearest Neighbor Interchange) is a postprocessing algorithm for phylogenetic trees. It swaps two distant 3-subtrees if the total tree length is reduced by doing so, until no such swaps are left. This algorithm is from Desper and Gascuel, *Fast and Accurate Phylogeny Reconstruction Algorithms Based on the Minimum-Evolution Principle*, 2002.

/graph drawing/no phylogenetic tree optimization=⟨*string*⟩

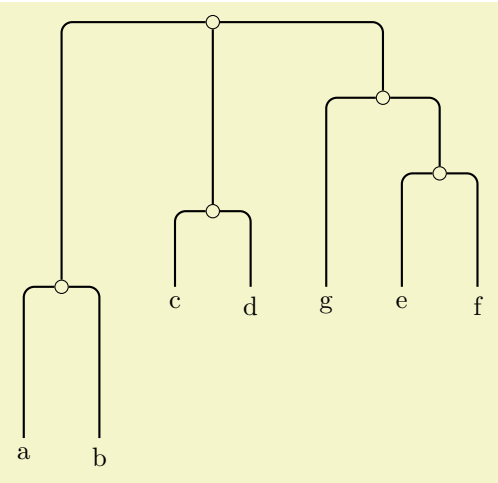
Switches off any phylogenetic tree optimization.

34.2 Laying out the Phylogram

/graph drawing/rooted rectangular phylogram=⟨*string*⟩

A rooted rectangular phylogram is... ...

Example



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout,
              rooted rectangular phylogram,
              balanced minimum evolution,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

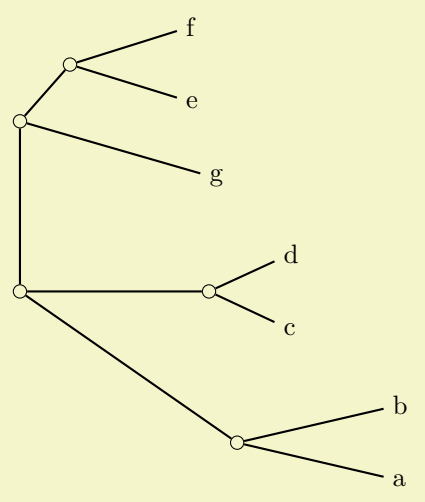
/graph drawing/rectangular phylogram=⟨*string*⟩

An alias for `rooted rectangular phylogram`

/graph drawing/rooted straight phylogram=⟨*string*⟩

A rooted straight phylogram is... ...

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout,
              rooted straight phylogram,
              balanced minimum evolution, grow=right,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

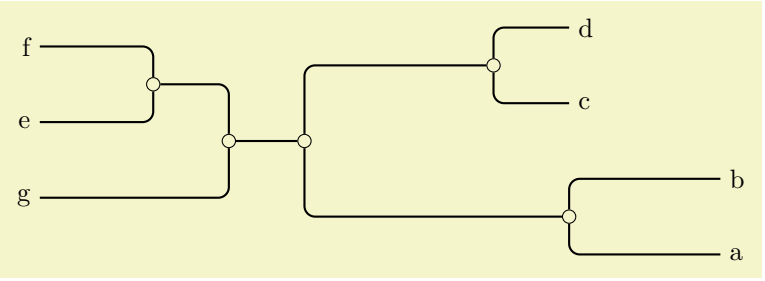
/graph drawing/straight phylogram=⟨*string*⟩

An alias for `rooted straight phylogram`

/graph drawing/unrooted rectangular phylogram=⟨*string*⟩

A unrooted rectangular phylogram is... ...

Example

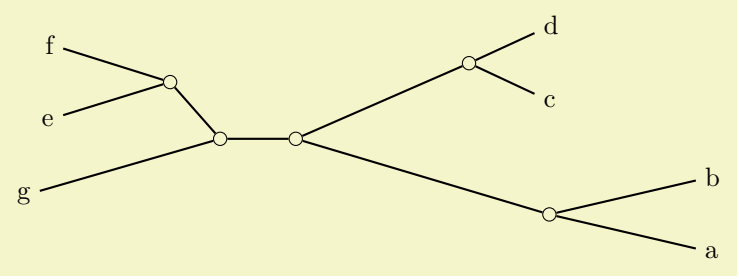



```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout,
              unrooted rectangular phylogram,
              balanced minimum evolution, grow=right,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

/graph drawing/unrooted straight phylogram=⟨*string*⟩

A unrooted straight phylogram is... ...

Example




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {phylogenetics}
\tikz \graph [phylogenetic tree layout,
              unrooted straight phylogram,
              balanced minimum evolution, grow=right,
              distance matrix={
                0 4 9 9 9 9 9
                4 0 9 9 9 9 9
                9 9 0 2 7 7 7
                9 9 2 0 7 7 7
                9 9 7 7 0 3 5
                9 9 7 7 3 0 5
                9 9 7 7 5 5 0}]
  { a, b, c, d, e, f, g };
```

/graph drawing/evolutionary unit length=⟨*length*⟩ (initially 1cm)

Specifies how long a "unit" of evolutionary time should be on paper. For instance, if two nodes in a phylogenetic tree have an evolutionary distance of 3 and this length is set to 1cm, then they will be 3cm apart in a straight-line phylogram. (This key used to be called distance scaling factor.)

35 Graph Drawing Algorithms: Edge Routing

by Till Tantau

Graph Drawing Library routing

```
\usegdlibrary{routing} % LaTeX and plain TeX
\usegdlibrary[routing] % ConTeXt
```

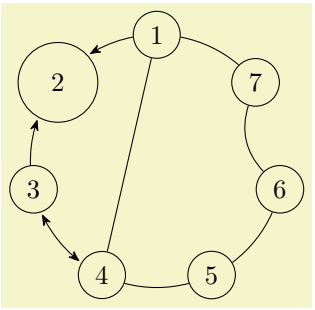
This library contains algorithms for routing edges through a graph.

`/graph drawing/necklace routing=`⟨*string*⟩

Bends all edges of a graph that lie on "necklaces" along these necklaces. Some graph drawing algorithms lay out some or all nodes along a path, which is then called a *necklace*. For instance, the `simple necklace layout` places all nodes on a circle and that circle is the "necklace". When the `necklace routing` edge routing algorithm is selected, all edges that connect subsequent nodes on such a necklace are bend in such a way that the "follow the necklace path". In the example case, this will cause all edges that connect adjacent nodes to become arcs on of the circle on which the nodes lie.

Note that local edge routing options for an edge may overrule the edge routing computed by the algorithm as in the edge from 6 to 7 in the example.

Example




```
\usetikzlibrary {arrows.meta,bending,graphs,graphdrawing} \usegdlibrary {circular,routing}
\tikz \graph [simple necklace layout, node distance=1.5cm,
              necklace routing,
              nodes={draw,circle}, edges={>={Stealth[round,sep,bend]}}]
  { 1 -> 2 [minimum size=30pt] <- 3 <-> 4 --
    5 -- 6 -- [bend left] 7 -- 1 -- 4 };
```

36 The Algorithm Layer

by Till Tantau

36.1 Overview

The present section is addressed at readers interested in implementing new graph drawing algorithms for the graph drawing system. Obviously, in order to do so, you need to have an algorithm in mind and also some programming skills; but fortunately only in the Lua programming language: Even though the graph drawing system was originally developed as an extension of Ti*k*Z, is has been restructured so that the "algorithm layer" where you define algorithms is scrupulously separated from Ti*k*Z. In particular, an algorithm declared and implemented on this layer can be used in with every "display layers", see Section 38, without change. Nevertheless, in the following we will use the Ti*k*Z display layer and syntax in our examples.

Normally, new graph drawing algorithms can and must be implemented in the Lua programming language, which is a small, easy-to-learn (and quite beautiful) language integrated into current versions of TEX. However, as explained in Section 37, you can also implement algorithms in C or C++ (and, possibly, in the future also in other languages), but this comes at a great cost concerning portability. In the present section, I assume that you are only interested in writing an algorithm using Lua.

In the following, after a small "hello world" example of graph drawing and a discussion of technical details like how to name files so that TEX will find them, we have a look at the main parts of the algorithm layer:

- Section 36.3 gives and overview of the available namespaces and also of naming conventions used in the graph drawing system.
- Section 36.4 explores what graph drawing scopes "look like on the algorithm layer". As the graph of a graph drawing scope is being parsed on the display layer, a lot of information is gathered: The nodes and edges of the graph are identified and the object-oriented model is built, but other information is also collected. For instance, a sequence of *events* is created during the parsing process. As another example, numerous kinds of *collections* may be identified by the parser. The parsed graph together with the event sequence and the collections are all gathered in a single table, called the *scope table* of the current graph drawing scope. Algorithms can access this table to retrieve information that goes beyond the "pure" graph model.

  One entry in this table is of particular importance: The *syntactic digraph*. While most graph drawing algorithms are not really interested in the "details" of how a graph was specified, for some algorithms it makes a big difference whether you write `a -> b` or `b <- a` in your specification of the graph. These algorithms can access the "fine details" of how the input graph was specified through the syntactic digraph; all other algorithms can access their `digraph` or `ugraph` fields and do not have to worry about the difference between `a -> b` and `b <- a`.
- Section 36.5 explains the object-oriented model of graphs used throughout the graph drawing system. Graph drawing algorithms do not get the "raw" specification used by the user to specify a graph (like `{a -> {b,c}}` in the `graph` syntax). Instead, what a graph drawing algorithm sees is "just" a graph object that provides methods for accessing the vertices and arcs.
- Section 36.6 explains how the information in the graph drawing scope is processed. One might expect that we simply run the algorithm selected by the user; however, things are more involved in practice. When the layout of a graph needs to be computed, only very few algorithms will actually be able to compute positions for the nodes of *every* graph. For instance, most algorithms implicitly assume that the input graph is connected; algorithms for computing layouts for trees assume that the input is, well, a tree; and so on. For this reason, graph drawing algorithms will not actually need the original input graph as their input, but some *transformed* version of it. Indeed, *all* graph drawing algorithms are treated as graph transformations by the graph drawing engine.

  This section explains how transformations are chosen and which transformations are applied by default.
- Section 36.7 documents the interface-to-algorithm class. This interface encapsulates all that an algorithm "sees" of the graph drawing system (apart from the classes in `model` and `lib`).
- Section 36.8 provides a number of complete examples that show how graph drawing algorithms can, actually, be implemented.

- Section 36.9 documents the different libraries functions that come with the graph drawing engine. For instance, there are library functions for computing the (path) distance of nodes in a graph; a parameter that is needed by some algorithms.

36.2 Getting Started

In this section, a "hello world" example of a graph drawing algorithm is given, followed by an overview of the organization of the whole engine.

36.2.1 The Hello World of Graph Drawing

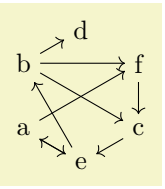
Let us start our tour of the algorithm layer with a "hello world" version of graph drawing: An algorithm that simply places all nodes of a graph in a circle of a fixed radius. Naturally, this is not a particularly impressive or intelligent graph drawing algorithm; but neither is the classical "hello world"... Here is a minimal version of the needed code (this is not the typical way of formulating the code, but it is the shortest; we will have a look at the more standard and verbose way in a moment):

```
pgf.gd.interface.InterfaceToAlgorithms.declare {
  key = "very simple demo layout",
  algorithm = {
    run =
      function (self)
        local alpha = (2 * math.pi) / #self.ugraph.vertices
        for i,vertex in ipairs(self.ugraph.vertices) do
          vertex.pos.x = math.cos(i * alpha) * 25
          vertex.pos.y = math.sin(i * alpha) * 25
        end
      end
  }
}
```

This code *declares* a new algorithm (`very simple demo layout`) and includes an implementation of the algorithm (through the `run` field of the `algorithm` field). When the `run` method is called, the `self` parameter will contain the to-be-drawn graph in its `ugraph` field. It is now the job of the code to modify the positions of the vertices in this graph (in the example, this is done by assigning values to `vertex.pos.x` and `vertex.pos.y`).

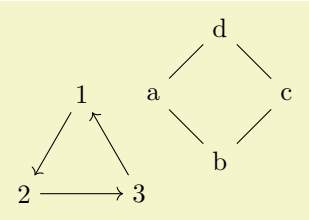
In order to actually *use* the algorithm, the above code first needs to be executed somehow. For Ti*k*Z, one can just call `\directlua` on it or put it in a file and then use `\directlua` plus `require` (a better alternative) or you put it in a file like `simpledemo.lua` and use `\usegdlibrary{simpledemo}` (undoubtedly the "best" way). For another display layer, like a graphical editor, the code could also be executed through the use of `require`.

Executing the code "just" declares the algorithm, this is what the `declare` function does. Inside some internal tables, the algorithm layer will store the fact that a `very simple demo layout` is now available. The algorithm layer will also communicate with the display layer through the binding layer to advertise this fact to the "user". In the case of Ti*k*Z, this means that the option key `very simple demo layout` becomes available at this point and we can use it like this:




```
\tikz [very simple demo layout]
  \graph { f -> c -> e -> a -> {b -> {c, d, f}, e -> b}};
```

It turns out, that our little algorithm is already more powerful than one might expect. Consider the following example:




```
\tikz [very simple demo layout, componentwise]
  \graph {
    1 -> 2 ->[orient=right] 3 -> 1;
    a -- b --[orient=45]  c -- d -- a;
  };
```

Note that, in our algorithm, we "just" put all nodes on a circle around the origin. Nevertheless, the graph gets decomposed into two connected components, the components are rotated so that the edge from node `2` to node `3` goes from left to right and the edge from `b` to `c` goes up at an angle of 45°, and the components are placed next to each other so that some spacing is achieved.

The "magic" that achieves all this behind the scenes is called "graph transformations". They will heavily pre- and postprocess the input and output of graph drawing algorithms to achieve the above results.

Naturally, some algorithms may not wish their inputs and/or outputs to be "tampered" with. An algorithm can easily configure which transformations should be applied, by passing appropriate options to `declare`.

36.2.2 Declaring an Algorithm

Let us now have a look at how one would "really" implement the example algorithm. First of all, we place our algorithm in a separate file called, say, `ExampleLayout.lua`. This way, by putting it in a separate file, all display layers can easily install the algorithm at runtime by saying `require "ExampleLayout"`.

Next, the `declare` function is needed quite often, so it makes sense to create a short local name for it:

```
-- This is the file ExampleLayout.lua
local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare
```

The `declare` function is the work-horse of the algorithm layer. It takes a table that contains at least a `key` field, which must be a unique string, and some other fields that specify in more detail what kind of key is declared. Once declared through a call of `declare`, the "key" can be used on the display layer.

For declaring an algorithm, the table passed to `declare` must contain a field `algorithm`. This field, in turn, must (normally) be set to a table that will become the algorithm class. In the above example, our algorithm was so simple that we could place the whole definition of the class inside the call of `declare`, but normally the class is defined in more detail after the call to `declare`:

```
local ExampleClass = {}  -- A local variable holding the class table

declare {
  key = "very simple demo layout",
  algorithm = ExampleClass
}

function ExampleClass:run ()
  local alpha = (2 * math.pi) / #self.ugraph.vertices
  ...
end
```

The effect of the `declare` will be that the table stored in `ExampleClass` is setup to form a class in the sense of object-oriented programming. In particular, a static `new` function is installed.

Now, whenever the user uses the key `very simple demo layout` on a graph, at some point the graph drawing engine will create a new instance of the `ExampleClass` using `new` and will then call the `run` method of this class. The class can have any number of other methods, but `new` and `run` are the only ones directly called by the graph drawing system.

36.2.3 The Run Method

The `run` method of an algorithm classes lies at the heart of any graph drawing algorithm. This method will be called whenever a graph needs to be laid out. Upon this call, the `self` object will have some important fields set:

- `ugraph` This stands for "undirected graph" and is the "undirected" version of the to-be-laid out graph. In this graph, whenever there is an arc between u and v, there is also an arc between v and u. It is

obtained by considering the syntactic digraph and then "forgetting" about the actual direction of the edges.

When you have set certain `preconditions` in your algorithm class, like `connected=true`, the `ugraph` will satisfy these conditions. In particular, the `ugraph` typically will not be the underlying undirected graph of the complete syntactic digraph, but rather of some part of it. The use of (sub)layouts will also modify the syntactic digraph is fancy ways.

Refer to this graph whenever your algorithm is "most comfortable" with an undirected graph, as is the case for instance for most force-base algorithms.

- `digraph` This stands for "directed graph" and is the "semantically directed" version of the to-be-laid out graph. Basically, when happens is that reverse edges in the syntactic digraph (an edge like `b <- a`) will yield an `Arc` from `a` to `b` in the `digraph` while they yield a `b` to `a` arc and edge in the syntactic digraph. Also, undirected edges like `a -- b` are replaced by directed edges in both directions between the vertices.
- `scope` The graph drawing scope.
- `layout` The layout object for this graph. This is a collection of kind `layout`.

36.2.4 Loading Algorithms on Demand

In order to use the `very simple demo layout` on the display layer, `declare` must have been called for this key. However, we just saw that the `declare` function takes the actual class table as parameter and, thus, whenever an algorithm is declared, it is also completely loaded and compiled at this point.

This is not always desirable. A user may wish to include a number of libraries in order to declare a large number of potentially useful algorithms, but will not actually use all of them. Indeed, at least for large, complex algorithms, it is preferable that the algorithm's code is loaded only when the algorithm is used for the first time.

Such a "loading of algorithms on demand" is supported through the option of setting the `algorithm` field in a `declare` to a string. This string must now be the file name of a Lua file that contains the code of the actual algorithm. When the key is actually used for the first time, this file will be loaded. It must return a table that will be plugged into the `algorithm` field; so subsequent usages of the key will not load the file again.

The net effect of all this is that you can place implementations of algorithms in files separate from interface files that just contain the `declare` commands for these algorithms. You will typically do this only for rather large algorithms.

For our example, the code would look like this:

```
-- File ExampleLayout.lua
local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare
declare {
  key = "very simple demo layout",
  algorithm = "ExampleLayoutImplementation"
}
```

```
-- File ExampleLayoutImplementation.lua
local ExampleClass = {}
function ExampleClass:run ()
  local alpha = (2 * math.pi) / #self.ugraph.vertices
  ...
end
return ExampleClass
```

36.2.5 Declaring Options

Let us now make our example algorithm a bit more "configurable". For this, we use `declare` once more, but instead of the `algorithm` field, we use a `type` field. This tells the display layer that the key is not used to select an algorithm, but to configure "something" about the graph or about nodes or edges.

In our example, we may wish to configure the radius of the graph. So, we introduce a `radius` key (actually, this key already exists, so we would not need to declare it, but let us do so anyway for example purposes):

```
declare {
  key = "radius",
  type = "length",
  initial = "25pt"
}
```

This tells the display layer that there is now an option called `radius`, that users set it to some "length", and that if it is not set at all, then the 25pt should be used.

To access what the user has specified for this key, an algorithm can access the `options` field of a graph, vertex, or arc at the key's name:

```
          vertex.pos.x = math.cos(i * alpha) * vertex.options.radius
          vertex.pos.y = math.sin(i * alpha) * vertex.options.radius
```

36.2.6 Adding Inline Documentation

You should always document the keys you `declare`. For this, the `declare` function allows you to add three fields to its argument table:

- `summary` This should be a string that succinctly summarizes the effect this key has. The idea is that this text will be shown as a "tooltip" in a graphical editor or will be printed out by a command line tool when a user requests help about the key. You can profit from using Lua's `[[` and `]]` syntax for specifying multi-line strings.

  Also, when the file containing the key is parsed for this manual, this text will be shown.

- `documentation` When present, this field contains a more extensive documentation of the key. It will also be shown in this manual, but typically not as a tool tip.

- `examples` This should either be a single string or an array of strings. Each string should be an example demonstrating how the key is used in Ti*k*Z. They will all be included in the manual, each surrounded by a `codeexample` environment.

Let us augment our `radius` key with some documentation. The three dashes before the `declare` are only needed when the declaration is part of this manual and they will trigger an inclusion of the key in the manual.

```
---
declare {
  key = "radius",
  type = "length",
  initial = "25pt",
  summary = [[
    Specifies the radius of a circle on which the nodes are placed when
    the |very simple example layout| is used. Each vertex can have a
    different radius.
  ]],
  examples = [[
    \tikz \graph [very simple example layout, radius=2cm] {
      a -- b -- c -- d -- e;
    };
  ]]
}
```

As a courtesy, all of the strings given in the documentation can start and end with quotation marks, which will be removed. (This helps syntax highlighting with editors that do not recognize the `[[` to `]]` syntax.) Also, the indentation of the strings is removed (we compute the minimum number of leading spaces on any line and remove this many spaces from all lines).

36.2.7 Adding External Documentation

As an alternative to inlining documentation, you can also store the documentation of keys in a separate file that is loaded only when the documentation is actually accessed. Since this happens only rarely (for instance, not at all, when Ti*k*Z is run, except for this manual), this will save time and space. Also, for C code, it is impractical to store multi-line documentation strings directly in the C file.

In order to store documentation externally, instead of the `summary`, `documentation`, and `examples` keys, you provide the key `documentation_in`. The `documentation_in` key must be set to a string that is input using `require`.

In detail, when someone tries to access the `summary`, `documentation`, or `examples` field of a key and these keys are not (yet) defined, the system checks whether the `documentation_in` key is set. If so, we apply `require` to the string stored in this field. The file loaded in this way can now setup the missing fields of the current key and, typically, also of all other keys defined in the same file as the current key. For this purpose, it is advisable to use the `pgf.gd.doc` class:

Lua table `doc` (declared in `pgf.gd.doc`)

The table doc is used for documentation purposes. It is used to provide lazy documentation for keys, that is, to install documentation for keys only when this information is requested and when the documentation is kept in a separate file.

Using the doc facility is easy:

1. In the `declare` statement of the key, you do not provide fields like `documentation` or `summary`. Rather, you provide the field `documentation_in`, which gets the name of a Lua file the will be read whenever one of the fields `documentation`, `summary`, or `examples` is requested for the key.
2. When the key is requested, `require` will be applied to the filename given in the `documentation_in` field.
3. In this file, you start with the following code:

```
local doc           = require 'pgf.gd.doc'
local key           = doc.key
local documentation = doc.documentation
local summary       = doc.summary
local example       = doc.example
```

   This will setup nice shortcuts for the commands you are going to use in your file.
4. Next, for each to-be-lazily-documented key, add a block to the file like the following:

```
---
key           "my radius"
summary       "This key specifies a radius."
documentation
[[
This key is used, whenever...
]]
example       "\tikz \graph [foo layout, my radius=5] { a--b };"
example       "\tikz \graph [foo layout, my radius=3] { c--d };"
```

   Note that `[[` and `]]` are used in Lua for raw multi-line strings.

   The effect of the above code will be that for the key `my radius` the different field like `summary` or `documentation` get updated. The `key` function simple "selects" a key and subsequent commands like `summary` will update this key until a different key is selected through another use of `key`.

Alphabetical method summary:

`function doc.documentation (string)`

`function doc.example (input)`

`function doc.key (key)`

`function doc.summary (string)`

`function doc.key(key)`

Selects the key which will be subsequently updated by the other functions of this class.

Parameters: 1. `key` A key.

`function doc.summary(string)`

Updates (replaces) the summary field of the last key selected through the `key` command.

Parameters: 1. `string` A (new) summary string.

`function doc.documentation(string)`

Updates (replaces) the documentation field of the last key selected through the `key` command.

Parameters: 1. `string` A (new) documentation string. Typically, the `[[` syntax will be used to specify this string.

`function doc.example(input)`

Adds an example to the `examples` field of the last key selected through the `key` command.

Parameters: 1. `input` Either a string of example or a table with two fields: ”code” and ”options”.

As a longer example, consider the following declarations:

```
---
declare {
  key               = "very simple demo layout",
  algorithm         = ExampleClass,
  documentation_in  = "documentation_file"
}

---
declare {
  key               = "radius",
  type              = "length",
  initial           = "25",
  documentation_in  = "documentation_file"
}
```

The file `documentation_file.lua` would look like this:

```
-- File documentation_file.lua
local key           = require 'pgf.gd.doc'.key
local documentation = require 'pgf.gd.doc'.documentation
local summary       = require 'pgf.gd.doc'.summary
local example       = require 'pgf.gd.doc'.example

key           "very simple demo layout"
documentation "This layout is a very simple layout that, ..."

key           "radius"
summary       "Specifies the radius of a circle on which the nodes are placed."
documentation
[[
This key can be used together with |very simple example layout|. An
important feature is that...
]]
example
[[
\tikz \graph [very simple example layout, radius=2cm]
{ a -- b -- c -- d -- e; };
]]
```

36.3 Namespaces and File Names

36.3.1 Namespaces

All parts of the `graphdrawing` library reside in the Lua “namespace” `pgf.gd`, which is itself a “sub-namespace” of `pgf`. For your own algorithms, you are free to place them in whatever namespace you like; only for the official distribution of PGF everything has been put into the correct namespace.

Let us now have a more detailed look at these namespaces. A namespace is just a Lua table, and sub-namespaces are just subtables of namespace tables. Following the Java convention, namespaces are in lowercase letters. The following namespaces are part of the core of the graph drawing engine:

- `pgf` This namespace is the main namespace of PGF. Other parts of PGF and Ti*k*Z that also employ Lua should put an entry into this table. Since, currently, only the graph drawing engine adheres to this rule, this namespace is declared inside the graph drawing directory, but this will change.

  The `pgf` table is the *only* entry into the global table of Lua generated by the graph drawing engine (or, PGF, for that matter). If you intend to extend the graph drawing engine, do not even *think* of polluting the global namespace. You will be fined.

- `pgf.gd` This namespace is the main namespace of the graph drawing engine, including the object-oriented models of graphs and the layout pipeline. Algorithms that are part of the distribution are also inside this namespace, but if you write your own algorithms you do not need place them inside this namespace. (Indeed, you probably should not before they are made part of the official distribution.)

- `pgf.gd.interface` This namespace handles, on the one hand, the communication between the algorithm layer and the binding layer and, on the other hand, the communication between the display layer (Ti*k*Z) and the binding layer.
- `pgf.gd.binding` So-called "bindings" between display layers and the graph drawing system reside in this namespace.
- `pgf.gd.lib` Numerous useful classes that "make an algorithm's your life easier" are collected in this namespace. Examples are a class for decomposing a graph into connected components or a class for computing the ideal distance between two sibling nodes in a tree, taking all sorts of rotations and separation parameters into account.
- `pgf.gd.model` This namespace contains all Lua classes that are part of the object-oriented model of graphs employed throughout the graph drawing engine. For readers familiar with the model–view–controller pattern: This is the namespace containing the model-part of this pattern.
- `pgf.gd.control` This namespace contains the "control logic" of the graph drawing system. It will transform graphs according to rules, disassemble layouts and sublayouts and will call the appropriate algorithms. For readers still familiar with the model–view–controller pattern: This is the namespace containing the control-part of this pattern.
- `pgf.gd.trees` This namespace contains classes that are useful for dealing with graphs that are trees. In particular, it contains a class for computing a spanning tree of an arbitrary connected graph; an operation that is an important preprocessing step for many algorithms.

  In addition to providing "utility functions for trees", the namespace *also* includes actual algorithms for computing graph layouts like `pgf.gd.trees.ReingoldTilford1981`. It may seem to be a bit of an "impurity" that a namespace mixes utility classes and "real" algorithms, but experience has shown that it is better to keep things together in this way.

  Concluding the analogy to the model–view–controller pattern, a graph drawing algorithm is, in a loose sense, the "view" part of the pattern.
- `pgf.gd.layered` This namespace provides classes and functions for "layered" layouts; the Sugiyama layout method being the most well-known one. Again, the namespace contains both algorithms to be used by a user and utility functions.
- `pgf.gd.force` Collects force-based algorithms and, again, also utility functions and classes.
- `pgf.gd.examples` Contains some example algorithms. They are *not* intended to be used directly, rather they should serve as inspirations for readers wishing to implement their own algorithms.

There are further namespaces that also reside in the `pgf.gd` namespace, these namespaces are used to organize different graph drawing algorithms into categories.

In Lua, similarly to Java, when a class `SomeClass` is part of, say, the namespace `pgf.gd.example`, it is customary to put the class's code in a file `SomeClass.lua` and then put this class in a directory `example`, that is a subdirectory of a directory `gd`, which is in turn a subdirectory of a directory `pgf`. When you write `require "pgf.gd.example.SomeClass"` the so-called *loader* will turn this into a request for the file `pgf/gd/example/SomeClass.lua` (for Unix systems).

36.3.2 Defining and Using Namespaces and Classes

There are a number of rules concerning the structure and naming of namespaces as well as the naming of files. Let us start with the rules for naming namespaces, classes, and functions. They follow the "Java convention":

1. A namespace is a short lowercase `word`.
2. A function in a namespace is in `lowercase_with_underscores_between_words`.
3. A class name is in `CamelCaseWithAnUppercaseFirstLetter`.
4. A class method name is in `camelCaseWithALowercaseFirstLetter`.

From Lua's point of view, every namespace and every class is just a table. However, since these tables will be loaded using Lua's `require` function, each namespace and each class must be placed inside a separate file (unless you modify the `package.loaded` table, but, then, you know what you are doing anyway). Inside such a file, you should first declare a local variable whose name is the name of the namespace or class that you intend to define and then assign a (possibly empty) table to this variable:

```
-- File pgf.gd.example.SomeClass.lua:
local SomeClass = {}
```

Next, you should add your class to the encompassing namespace. This is achieved as follows:

```
require("pgf.gd.example").SomeClass = SomeClass
```

The reason this works is that the `require` will return the table that is the namespace `pgf.gd.example`. So, inside this namespace, the `SomeClass` field will be filled with the table stored in the local variable of the same name – which happens to be the table representing the class.

At the end of the file, you must write

```
return SomeClass
```

This ensures that the table that is defined in this file gets stored by Lua in the right places. Note that you need and should not use Lua's `module` command. The reason is that this command has disappeared in the new version of Lua and that it is not really needed.

Users of your class can import and use your class by writing:

```
...
local SomeClass = require "pgf.gd.examples.SomeClass"
...
```

36.4 The Graph Drawing Scope

Lua table Scope (declared in `pgf.gd.interface.Scope`)

In theory, graph drawing algorithms take graphs as input and output graphs embedded into the plane as output. In practice, however, the input to a graph drawing algorithm is not "just" the graph. Rather, additional information about the graph, in particular about the way the user specified the graph, is also important to many graph drawing algorithms.

The graph drawing system gathers both the original input graph as well as all additional information that is provided in the graph drawing scope inside a scope table. The object has a number of fields that inform an algorithm about the input.

For each graph drawing scope, a new `Scope` object is created. Graph drawing scopes are kept track of using a stack, but only the top of this stack is available to the interface classes.

Field syntactic_digraph

The syntactic digraph is a digraph that faithfully encodes the way the input graph is represented syntactically. However, this does not mean that the syntactic digraph contains the actual textual representation of the input graph. Rather, when an edge is specified as, say, `a <- b`, the syntactic digraph will contains an arc from `a` to `b` with an edge object attached to it that is labeled as a "backward" edge. Similarly, an edge `a -- b` is also stored as a directed arc from `a` to `b` with the label `--` attached to it. Algorithms will often be more interested graphs derived from the syntactic digraph such as its underlying undirected graph. These derived graphs are made accessible by the graph drawing engine during the preprocessing.

Field events

An array of `Event` objects. These objects, see the `Event` class for details, are created during the parsing of the input graph.

Field node_names

A table that maps the names of nodes to node objects. Every node must have a unique name.

Field coroutine

A Lua coroutine that is used internally to allow callbacks to the display layer to be issued deep down during a run of an algorithm.

Field `collections`

The collections specified inside the scope, see the `Collection` class.

Alphabetical method summary:

`function Scope.new (initial)`

`function Scope.new(initial)`

Create a new `Scope` object.

Parameters: 1. `initial` A table of initial values for the newly created `Scope` object.

Returns: 1. The new scope object.

36.5 The Model Classes

All that a graph drawing algorithm will "see" of the graph specified by the user is a "graph object". Such an object is an object-oriented model of the user's graph that no longer encodes the specific way in which the user specified the graph; it only encodes which nodes and edges are present. For instance, the Ti*k*Z graph specification

```
graph { a -- {b, c} }
```

and the graph specification

```
node (a) { a }
child { node (b) {b} }
child { node (c) {c} }
```

will generate exactly the same graph object.

Lua namespace `pgf.gd.model`

This namespace contains the classes modeling graphs, nodes, and edges. Also, the `Coordinate` class is found here, since coordinates are also part of the modeling.

36.5.1 Directed Graphs (Digraphs)

Inside the graph drawing engine, the only model of a graph that is available treats graphs as

1. directed (all edges have a designated head and a designated tail) and
2. simple (there can be at most one edge between any pair of nodes).

These two properties may appear to be somewhat at odds with what users can specify as graphs and with what some graph drawing algorithms might expect as input. For instance, suppose a user writes

```
graph { a -- b --[red] c, b --[green, bend right] c }
```

In this case, it seems that the input graph for a graph drawing algorithm should actually be an *undirected* graph in which there are *multiple* edges (namely 2) between `b` and `c`. Nevertheless, the graph drawing engine will turn the user's input a directed simple graph in ways described later. You do not need to worry that information gets lost during this process: The *syntactic digraph,* which is available to graph drawing algorithms on request, stores all the information about which edges are present in the original input graph.

The main reasons for only considering directed, simple graphs are speed and simplicity: The implementation of these graphs has been optimized so that all operations on these graphs have a guaranteed running time that is small in practice.

Lua table `Digraph` (declared in `pgf.gd.model.Digraph`)

Each `Digraph` instance models a *directed, simple* graph. "Directed" means that all edges "point" from a head node to a tail node. "Simple" means that between any nodes there can be (at most) one edge. Since these properties are a bit at odds with the normal behavior of "nodes" and "edges" in Ti*k*Z, different names are used for them inside the `model` namespace: The class modeling "edges" is actually called `Arc` to stress that an arc has a specific "start" (the tail) and a specific "end" (the head). The class modeling "nodes" is actually called `Vertex`, just to stress that this is not a direct model of a Ti*k*Z

`node`, but can represent a arbitrary vertex of a graph, independently of whether it is an actual `node` in Ti*k*Z.

Time Bounds. Since digraphs are constantly created and modified inside the graph drawing engine, some care was taken to ensure that all operations work as quickly as possible. In particular:

- Adding an array of k vertices using the `add` method needs time $O(k)$.
- Adding an arc between two vertices needs time $O(1)$.
- Accessing both the `vertices` and the `arcs` fields takes time $O(1)$, provided only the above operations are used.

Deleting vertices and arcs takes more time:

- Deleting the vertices given in an array of k vertices from a graph with n vertices takes time $O(\max\{n, c\})$ where c is the number of arcs between the to-be-deleted nodes and the remaining nodes. Note that this time bound in independent of k. In particular, it will be much faster to delete many vertices by once calling the `remove` function instead of calling it repeatedly.
- Deleting an arc takes time $O(t_o + h_i)$ where t_o is the number of outgoing arcs at the arc's tail and h_i is the number of incoming arcs at the arc's head. After a call to `disconnect`, the next use of the `arcs` field will take time $O(|V| + |E|)$, while subsequent accesses take time $O(1)$ – till the next use of `disconnect`. This means that once you start deleting arcs using `disconnect`, you should perform as many additional `disconnect`s as possible before accessing `arcs` one more.

Stability. The `vertices` field and the array returned by `Digraph:incoming` and `Digraph:outgoing` are *stable* in the following sense: The ordering of the elements when you use `ipairs` on the will be the ordering in which the vertices or arcs were added to the graph. Even when you remove a vertex or an arc, the ordering of the remaining elements stays the same.

Field `vertices`

This array contains the vertices that are part of the digraph. Internally, this array is an object of type `LookupTable`, but you can mostly treat it as if it were an array. In particular, you can iterate over its elements using `ipairs`, but you may not modify the array; use the `add` and `remove` methods, instead.

```
local g = Digraph.new {}

g:add { v1, v2 } -- Add vertices v1 and v2
g:remove { v2 }  -- Get rid of v2.

assert (g:contains(v1))
assert (not g:contains(v2))
```

It is important to note that although each digraph stores a `vertices` array, the elements in this array are not exclusive to the digraph: A vertex can be an element of any number of digraphs. Whether or not a vertex is an element of digraph is not stored in the vertex, only in the `vertices` array of the digraph. To test whether a digraph contains a specific node, use the `contains` method, which takes time $O(1)$ to perform the test (this is because, as mentioned earlier, the `vertices` array is actually a `LookupTable` and for each vertex `v` the field `vertices[v]` will be true if, and only if, `v` is an element of the `vertices` array).

Do not use `pairs(g.vertices)` because this may cause your graph drawing algorithm to produce different outputs on different runs.

A slightly annoying effect of vertices being able to belong to several graphs at the same time is that the set of arcs incident to a vertex is not a property of the vertex, but rather of the graph. In other words, to get a list of all arcs whose tail is a given vertex `v`, you cannot say something like `v.outgoings` or perhaps `v:getOutgoings()`. Rather, you have to say `g:outgoing(v)` to get this list:

```
for _,a in ipairs(g:outgoing(v)) do  -- g is a Digraph object.
  pgf.debug ("There is an arc leaving " .. tostring(v) ..
             " heading to " .. tostring(a.head))
end
```

Naturally, there is also a method `g:incoming()`.

To iterate over all arcs of a graph you can say:

```
for _,v in ipairs(g.vertices) do
  for _,a in ipairs(g:outgoing(v)) do
    ...
  end
end
```

However, it will often be more convenient and, in case the there are far less arcs than vertices, also faster to write

```
for _,a in ipairs(g.arcs) do
  ...
end
```

Field `arcs`

For any two vertices `t` and `h` of a graph, there may or may not be an arc from `t` to `h`. If this is the case, there is an `Arc` object that represents this arc. Note that, since `Digraph`s are always simple graphs, there can be at most one such object for every pair of vertices. However, you can store any information you like for an `Arc` through a `Storage`, see the `Storage` class for details. Each `Arc` for an edge of the syntactic digraph stores an array called `syntactic_edges` of all the multiple edges that are present in the user's input.

Unlike vertices, the arc objects of a graph are always local to a graph; an `Arc` object can never be part of two digraphs at the same time. For this reason, while for vertices it makes sense to create `Vertex` objects independently of any `Digraph` objects, it is not possible to instantiate an `Arc` directly: only the `Digraph` method `connect` is allowed to create new `Arc` objects and it will return any existing arcs instead of creating new ones, if there is already an arc present between two nodes.

The `arcs` field of a digraph contains a `LookupTable` of all arc objects present in the `Digraph`. Although you can access this field normally and use it in `ipairs` to iterate over all arcs of a graph, note that this array is actually "reconstructed lazily" whenever an arc is deleted from the graph. What happens is the following: As long as you just add arcs to a graph, the `arcs` array gets updated normally. However, when you remove an arc from a graph, the arc does not get removed from the `arcs` array (which would be an expensive operation). Instead, the `arcs` array is invalidated (internally set to `nil`), allowing us to perform a `disconnect` in time $O(1)$. The `arcs` array is then ignored until the next time it is accessed, for instance when a user says `ipairs(g.arcs)`. At this point, the `arcs` array is reconstructed by adding all arcs of all nodes to it.

The bottom line of the behavior of the `arcs` field is that (a) the ordering of the elements may change abruptly whenever you remove an arc from a graph and (b) performing k `disconnect` operations in sequence takes time $O(k)$, provided you do not access the `arcs` field between calls.

Field `syntactic_digraph`

is a reference to the syntactic digraph from which this graph stems ultimately. This may be a cyclic reference to the graph itself.

Field `options`

If present, it will be a table storing the options set for the syntactic digraph.

Alphabetical method summary:

`function Digraph.new (initial)`

`function Digraph:__tostring ()`

`function Digraph:add (array)`

`function Digraph:arc (tail, head)`

`function Digraph:collapse (collapse_vertices, collapse_vertex, vertex_fun, arc_fun)`

`function Digraph:connect (s, t)`

`function Digraph:contains (v)`

`function Digraph:disconnect (v, t)`

`function Digraph:expand (vertex, vertex_fun, arc_fun)`

```
function Digraph:incoming (v)
function Digraph:orderIncoming (v, vertices)
function Digraph:orderOutgoing (v, vertices)
function Digraph:outgoing (v)
function Digraph:reconnect (arc, tail, head)
function Digraph:remove (array)
function Digraph:sortIncoming (v, f)
function Digraph:sortOutgoing (v, f)
function Digraph:sync ()
```

`function Digraph.new(initial)`

Graphs are created using the `new` method, which takes a table of `initial` values as input (like most `new` methods in the graph drawing system). It is permissible that this table of initial values has a `vertices` field, in which case this array will be copied. In contrast, an `arcs` field in the table will be ignored – newly created graphs always have an empty arcs set. This means that writing `Digraph.new(g)` where `g` is a graph creates a new graph whose vertex set is the same as `g`'s, but where there are no edges:

```
local g = Digraph.new {}
g:add { v1, v2, v3 }
g:connect (v1, v2)

local h = Digraph.new (g)
assert (h:contains(v1))
assert (not h:arc(v1, v2))
```

To completely copy a graph, including all arcs, you have to write:

```
local h = Digraph.new (g)
for _,a in ipairs(g.arcs) do h:connect(a.tail, a.head) end
```

This operation takes time $O(1)$.

Parameters: 1. `initial` A table of initial values. It is permissible that this array contains a `vertices` field. In this case, this field must be an array and its entries must be nodes, which will be inserted. If initial has an `arcs` field, this field will be ignored. The table must contain a field `syntactic_digraph`, which should normally be the syntactic digraph of the graph, but may also be the string `"self"`, in which case it will be set to the newly created (syntactic) digraph.

Returns: 1. A newly-allocated digraph.

`function Digraph:add(array)`

Add vertices to a digraph.

This operation takes time $O(|\texttt{array}|)$.

Parameters: 1. `array` An array of to-be-added vertices.

`function Digraph:remove(array)`

Remove vertices from a digraph.

This operation removes an array of vertices from a graph. The operation takes time linear in the number of vertices, regardless of how many vertices are to be removed. Thus, it will be (much) faster to delete many vertices by first compiling them in an array and to then delete them using one call to this method.

This operation takes time $O(\max\{|\texttt{array}|, |\texttt{self.vertices}|\})$.

Parameters: 1. `array` The to-be-removed vertices.

`function Digraph:contains(v)`

Test, whether a graph contains a given vertex.

This operation takes time $O(1)$.

Parameters: 1. `v` The vertex to be tested.

`function Digraph:arc(tail, head)`

Returns the arc between two nodes, provided it exists. Otherwise, `nil` is returned.

This operation takes time $O(1)$.

Parameters: *1.* `tail` The tail vertex *2.* `head` The head vertex

Returns: *1.* The arc object connecting them

`function Digraph:outgoing(v)`

Returns an array containing the outgoing arcs of a vertex. You may only iterate over his array using ipairs, not using pairs.

This operation takes time $O(1)$.

Parameters: *1.* `v` The vertex

Returns: *1.* An array of all outgoing arcs of this vertex (all arcs whose tail is the vertex)

`function Digraph:sortOutgoing(v, f)`

Sorts the array of outgoing arcs of a vertex. This allows you to later iterate over the outgoing arcs in a specific order.

This operation takes time $O(|\texttt{outgoing}| \log |\texttt{outgoings}|)$.

Parameters: *1.* `v` The vertex *2.* `f` A comparison function that is passed to `table.sort`

`function Digraph:orderOutgoing(v, vertices)`

Reorders the array of outgoing arcs of a vertex. The parameter array *must* contain the same set of vertices as the outgoing array, but possibly in a different order.

This operation takes time $O(|\texttt{outgoing}|)$, where `outgoing` is the array of `v`'s outgoing arcs in `self`.

Parameters: *1.* `v` The vertex *2.* `vertices` An array containing the outgoing vertices in some order.

`function Digraph:incoming(v)`

See `outgoing`.

`function Digraph:sortIncoming(v, f)`

See `sortOutgoing`.

`function Digraph:orderIncoming(v, vertices)`

See `orderOutgoing`.

`function Digraph:connect(s, t)`

Connects two nodes by an arc and returns the newly created arc object. If they are already connected, the existing arc is returned.

This operation takes time $O(1)$.

Parameters: *1.* `s` The tail vertex *2.* `t` The head vertex (may be identical to `tail` in case of a loop)

Returns: *1.* The arc object connecting them (either newly created or already existing)

`function Digraph:disconnect(v, t)`

Disconnect either a single vertex `v` from all its neighbors (remove all incoming and outgoing arcs of this vertex) or, in case two nodes are given as parameter, remove the arc between them, if it exists.

This operation takes time $O(|I_v| + |I_t|)$, where I_x is the set of vertices incident to x, to remove the single arc between v and v. For a single vertex v, it takes time $O(\sum_{y:\text{there is some arc between } v \text{ and } y \text{ or } y \text{ and } v} |I_y|)$.

Parameters: *1.* `v` The single vertex or the tail vertex *2.* `t` The head vertex

`function Digraph:reconnect(arc, tail, head)`

An arc is changed so that instead of connecting `self.tail` and `self.head`, it now connects a new `head` and `tail`. The difference to first disconnecting and then reconnecting is that all fields of the arc (other than `head` and `tail`, of course), will be "moved along". Reconnecting an arc in the same way as before has no effect.

If there is already an arc at the new position, fields of the to-be-reconnected arc overwrite fields of the original arc. This is especially dangerous with a syntactic digraph, so do not reconnect arcs of the syntactic digraph (which you should not do anyway).

The `arc` object may no longer be valid after a reconnect, but the operation returns the new arc object.

This operation needs the time of a disconnect (if necessary).

Parameters: *1.* `arc` The original arc object *2.* `tail` The new tail vertex *3.* `head` The new head vertex

Returns: *1.* The new arc object connecting them (either newly created or already existing)

`function Digraph:collapse(collapse_vertices, collapse_vertex, vertex_fun, arc_fun)`

Collapse a set of vertices into a single vertex

Often, algorithms will wish to treat a whole set of vertices "as a single vertex". The idea is that a new vertex is then inserted into the graph, and this vertex is connected to all vertices to which any of the original vertices used to be connected.

The `collapse` method takes an array of to-be-collapsed vertices as well as a vertex. First, it will store references to the to-be-collapsed vertices inside the vertex. Second, we iterate over all arcs of the to-be-collapsed vertices. If this arc connects a to-be-collapsed vertex with a not-to-be-collapsed vertex, the not-to-be-collapsed vertex is connected to the collapse vertex. Additionally, the arc is stored at the vertex.

Note that the collapse vertex will be added to the graph if it is not already an element. The collapsed vertices will not be removed from the graph, so you must remove them yourself, if necessary.

A collapse vertex will store the collapsed vertices so that you can call `expand` later on to "restore" the vertices and arcs that were saved during a collapse. This storage is *not* local to the graph in which the collapse occurred.

Parameters: *1.* `collapse_vertices` An array of to-be-collapsed vertices *2.* `collapse_vertex` The vertex that represents the collapse. If missing, a vertex will be created automatically and added to the graph. *3.* `vertex_fun` This function is called for each to-be-collapsed vertex. The parameters are the collapse vertex and the to-be-collapsed vertex. May be `nil`. *4.* `arc_fun` This function is called whenever a new arc is added between `rep` and some other vertex. The arguments are the new arc and the original arc. May be `nil`.

Returns: *1.* The new vertex that represents the collapsed vertices.

`function Digraph:expand(vertex, vertex_fun, arc_fun)`

Expand a previously collapsed vertex.

If you have collapsed a set of vertices in a graph using `collapse`, you can expand this set once more using this method. It will add all vertices that were previously removed from the graph and will also reinstall the deleted arcs. The collapse vertex is not removed.

Parameters: *1.* `vertex` A to-be-expanded vertex that was previously returned by `collapse`. *2.* `vertex_fun` A function that is called once for each reinserted vertex. The parameters are the collapse vertex and the reinstalled vertex. May be `nil`. *3.* `arc_fun` A function that is called once for each reinserted arc. The parameter is the arc and the `vertex`. May be `nil`.

`function Digraph:sync()`

Invokes the `sync` method for all arcs of the graph.

See also: `Arc:sync()`

`function Digraph:__tostring()`

Computes a string representation of this graph including all nodes and edges. The syntax of this representation is such that it can be used directly in Ti*k*Z's `graph` syntax.

Returns: *1.* `self` as string.

36.5.2 Vertices

Lua table `Vertex` (declared in `pgf.gd.model.Vertex`)

A `Vertex` instance models a node of graphs. Each `Vertex` object can be an element of any number of graphs (whereas an `Arc` object can only be an element of a single graph).

When a vertex is added to a digraph `g`, two tables are created in the vertex' storage: An array of incoming arcs (with respect to `g`) and an array of outgoing arcs (again, with respect to `g`). The fields are managed by the `Digraph` class and should not be modified directly.

Note that a `Vertex` is an abstraction of Ti*k*Z nodes; indeed the objective is to ensure that, in principle, we can use them independently of TEX. For this reason, you will not find any references to `tex` inside a `Vertex`; this information is only available in the syntactic digraph.

One important aspect of vertices are its anchors – a concept well familiar for users of Ti*k*Z, but since we need to abstract from Ti*k*Z, a separate anchor management is available inside the graph drawing system. It works as follows:

First of all, every vertex has a path, which is a (typically closed) line around the vertex. The display system will pass down the vertex' path to the graph drawing system and this path will be stored as a `Path` object in the `path` field of the vertex. This path lives in a special "local" coordinate system, that is, all coordinates of this path should actually be considered relative to the vertex' `pos` field. Note that the path is typically, but not always, "centered" on the origin. A graph drawing algorithm should arrange the vertices in such a way that the origins in the path coordinate systems are aligned.

To illustrate the difference between the origin and the vertex center, consider a tree drawing algorithm in which a node `root` has three children `a`, `b`, and `g`. Now, if we were to simply center these three letters vertically and arrange them in a line, the letters would appear to "jump up and down" since the height of the three letters are quite different. A solution is to shift the letters (and, thus, the paths of the vertices) in such a way that in all three letters the baseline of the letters is exactly at the origin. Now, when a graph drawing algorithm aligns these vertices along the origins, the letters will all have the same baseline.

Apart from the origin, there may be other positions in the path coordinate system that are of interest – such as the center of the vertex. As mentioned above, this need not be the origin and although a graph drawing algorithm should align the origins, *edges* between vertices should head toward these vertex centers rather that toward the origins. Other points of interest might be the "top" of the node.

All points of special interest are called "anchors". The `anchor` method allows you to retrieve them. By default, you always have access to the `center` anchor, but other anchors may or may not be available also, see the `anchor` method for details.

Field `pos`

A coordinate object that stores the position where the vertex should be placed on the canvas. The main objective of graph drawing algorithms is to update this coordinate.

Field `name`

An optional string that is used as a textual representation of the node.

Field `path`

The path of the vertex's shape. This is a path along the outer line resulting from stroking the vertex's original shape. For instance, if you have a quadratic shape of size 1cm and you stroke the path with a pen of 2mm thickness, this `path` field would store a path of a square of edge length 12mm.

Field `anchors`

A table of anchors (in the TikZ sense). The table is indexed by the anchor names (strings) and the values are `Coordinate`s. Currently, it is only guaranteed that the `center` anchor is present. Note that the `center` anchor need not lie at the origin: A graph drawing system should align nodes relative to the origin of the path's coordinate system. However, lines going to and from the node will head towards the `center` anchor. See Section 28.8 for details.

Field `options`

A table of options that contains user-defined options.

Field `animations`

An array of attribute animations for the node. When an algorithm adds entries to this array, the display layer should try to render these. The syntax is as follows: Each element in the array is a table with a field `attribute`, which must be a string like `"opacity"` or `"translate"`, a field `entries`, which must be an array to be explained in a moment, and field `options`, which must be a table of the same syntax as the `options` field. For the `entries` array, each element must be table with two field: `t` must be set to a number, representing a time in seconds, and `value`, which must

be set to a value that the `attribute` should have at the given time. The entries and the options will then be interpreted as described in PGF's basic layer animation system, except that where a `\pgfpoint` is expected you provide a `Coordinate` and a where a path is expected you provide a `Path`.

Field `shape`

A string describing the shape of the node (like `rectangle` or `circle`). Note, however, that this is more "informative"; the actual information that is used by the graph drawing system for determining the extent of a node, its bounding box, convex hull, and line intersections is the `path` field.

Field `kind`

A string describing the kind of the node. For instance, a node of type `"dummy"` does not correspond to any real node in the graph but is used by the graph drawing algorithm.

Field `event`

The `Event` when this vertex was created (may be `nil` if the vertex is not part of the syntactic digraph).

Field `incomings`

A table indexed by `Digraph` objects. For each digraph, the table entry is an array of all vertices from which there is an `Arc` to this vertex. This field is internal and may not only be accessed by the `Digraph` class.

Field `outgoings`

Like `incomings`, but for outgoing arcs.

Alphabetical method summary:

`function Vertex.new (values)`

`function Vertex:anchor (anchor)`

`function Vertex:boundingBox ()`

`function Vertex.new(values)`

Create a new vertex. The `initial` parameter allows you to setup some initial values.

Usage:

```
local v = Vertex.new { name = "hello", pos = Coordinate.new(1,1) }
```

Parameters: *1.* `initial` Values to override default node settings. The following are permissible:

`pos` Initial position of the node.
`name` The name of the node. It is optional to define this.
`path` A `Path` object representing the vertex's hull.
`anchors` A table of anchors.
`options` An options table for the vertex.
`animations` An array of generated animation attributes.
`shape` A string describing the shape. If not given, `"none"` is used.
`kind` A kind like `"node"` or `"dummy"`. If not given, `"dummy"` is used.

Returns: *1.* A newly allocated node.

`function Vertex:boundingBox()`

Returns a bounding box of a vertex.

Returns: *1.* `min_x` The minimum x value of the bounding box of the path

Returns: *1.* `min_y` The minimum y value

Returns: *1.* `max_x`

Returns: *1.* `max_y`

Returns: *1.* `center_x` The center of the bounding box

Returns: *1.* `center_y`

`function Vertex:anchor(anchor)`

Returns an anchor position in a vertex. First, we try to look the anchor up in the vertex's `anchors` table. If it is not found there, we test whether it is one of the direction strings `north`, `south east`, and so on. If so, we consider a line from the center of the node to the position on the bounding box that corresponds to the given direction (so `south east` would be the lower right corner). We intersect this line with the vertex's path and return the result. Finally, if the above fails, we try to consider the anchor as a number and return the intersection of a line starting at the vertex's center with the number as its angle and the path of the vertex.

Parameters: 1. `anchor` An anchor as detailed above

Returns: 1. A coordinate in the vertex's local coordinate system (so add the `pos` field to arrive at the actual position). If the anchor was not found, `nil` is returned

36.5.3 Arcs

Lua table `Arc` (declared in `pgf.gd.model.Arc`)

An arc is a light-weight object representing an arc from a vertex in a graph to another vertex. You may not create an `Arc` by yourself, which is why there is no `new` method, arc creation is done by the Digraph class.

Every arc belongs to exactly one graph. If you want the same arc in another graph, you need to newly connect two vertices in the other graph.

You may read the `head` and `tail` fields, but you may not write them. In order to store data for an arc, use `Storage` objects.

Between any two vertices of a graph there can be only one arc, so all digraphs are always simple graphs. However, in the specification of a graph (the syntactic digraph), there might be multiple edges between two vertices. This means, in particular, that an arc has no `options` field. Rather, it has several `optionsXxxx` functions, that will search for options in all of the syntactic edges that "belong" to an edge.

In order to *set* options of the edges, you can set the `generated_options` field of an arc (which is `nil` by default), see the `declare_parameter_sequence` function for the syntax. Similar to the `path` field below, the options set in this table are written back to the syntactic edges during a sync.

Finally, there is also an `animations` field, which, similarly to the `generated_options`, gets written back during a sync when it is not `nil`.

In detail, the following happens: Even though an arc has a `path`, `generated_options`, and `animations` fields, setting these fields does not immediately set the paths of the syntactic edges nor does it generate options. Indeed, you will normally want to setup and modify the `path` field of an arc during your algorithm and only at the very end, "write it back" to the multiple syntactic edges underlying the graph. For this purpose, the method `sync` is used, which is called automatically for the `ugraph` and `digraph` of a scope as well as for spanning trees.

The bottom line concerning the `path` field is the following: If you just want a straight line along an arc, just leave the field as it is (namely, `nil`). If you want to have all edges along a path to follow a certain path, set the `path` field of the arc to the path you desire (typically, using the `setPolylinePath` or a similar method). This will cause all syntactic edges underlying the arc to be set to the specified path. In the event that you want to set different paths for the edges underlying a single arc differently, set the `path` fields of these edges and set the `path` field of the arc to `nil`. This will disable the syncing for the arc and will cause the edge `paths` to remain untouched.

Field `tail`

The tail vertex of the arc.

Field `head`

The head vertex of the arc. May be the same as the tail in case of a loop.

Field `path`

If non-nil, the path of the arc. See the description above.

Field `generated_options`

If non-nil, some options to be passed back to the original syntactic edges, see the description above.

Field animations

If non-nil, some animations to be passed back to the original syntactic edges. See the description of the animations field for Vertex for details on the syntax.

Field syntactic_edges

In case this arc is an arc in the syntactic digraph (and only then), this field contains an array containing syntactic edges ("real" edges in the syntactic digraph) that underly this arc. Otherwise, the field will be empty or nil.

Alphabetical method summary:

function Arc:eventIndex ()

function Arc:headAnchorForArcPath ()

function Arc:optionsAccumulated (option, accumulator, only_aligned)

function Arc:optionsArray (option)

function Arc:options (option, only_aligned)

function Arc:pointCloud ()

function Arc:setPolylinePath (coordinates)

function Arc:spanPriority ()

function Arc:sync ()

function Arc:syntacticTailAndHead ()

function Arc:tailAnchorForArcPath ()

function Arc:optionsArray(option)

Get an array of options of the syntactic edges corresponding to an arc.

An arc in a digraph is typically (but not always) present because there are one or more edges in the syntactic digraph between the tail and the head of the arc or between the head and the tail.

Since for every arc there can be several edges present in the syntactic digraph, an option like length may have been given multiple times for the edges corresponding to the arc.

If your algorithm gets confused by multiple edges, try saying a:options(your_option). This will always give the "most sensible" choice of the option if there are multiple edges corresponding to the same arc.

Parameters: 1. option A string option like "length".

Returns: 1. A table with the following contents:

1. It is an array of all values the option has for edges corresponding to self in the syntactic digraph. Suppose, for instance, you write the following:

```
graph {
  tail -- [length=1] head,  % multi edge 1
  tail -- [length=3] head,  % mulit edge 2
  head -- [length=8] tail,  % multi edge 3
  tail --            head,  % multi edge 4
  head -- [length=7] tail,  % multi edge 5
  tail -- [length=2] head,  % multi edge 6
}
```

   Suppose, furthermore, that length has been setup as an edge option. Now suppose that a is the arc from the vertex tail to the vertex head. Calling a:optionsArray('length') will yield the array part {1,3,2,8,7}. The reason for the ordering is as follows: First come all values length had for syntactic edges going from self.tail to self.head in the order they appear in the graph description. Then come all values the options has for syntactic edges going from self.head to self.tail. The reason for this slightly strange behavior is that many algorithms do not really care whether someone writes a --[length=1] b or b --[length=1] a; in both cases they would "just" like to know that the length is 1.
2. There is field called aligned, which is an array storing the actual syntactic edge objects whose values can be found in the array part of the returned table. However, aligned contains only the syntactic edges pointing "in the same direction" as the arc, that is, the tail and head of the

syntactic edge are the same as those of the arc. In the above example, this array would contain the edges with the comment numbers `1`, `2`, and `6`.
Using the length of this array and the fact that the "aligned" values come first in the table, you can easily iterate over the `option`'s values of only those edges that are aligned with the arc:

```
local a = g:arc(tail.head)   -- some arc
local opt = a:optionsArray('length')
local sum = 0
for i=1,#opt.aligned do
  sum = sum + opt[i]
end
```

3. There is a field called `anti_aligned`, which is an array containing exactly the edges in the array part of the table not aligned with the arc. The numbering start at `1` as usual, so the ith entry of this table corresponds to the entry at position i + `#opt.aligned` of the table.

`function Arc:options(option, only_aligned)`

Returns the first option, that is, the first entry of `Arc:optionsArray(option)`. However, if the `only_aligned` parameter is set to true and there is no option with any aligned syntactic edge, `nil` is returned.

Parameters: *1.* `option` An option *2.* `only_aligned` If true, only aligned syntactic edges will be considered.

Returns: *1.* The first entry of the `optionsArray`

`function Arc:optionsAccumulated(option, accumulator, only_aligned)`

Get an accumulated value of an option of the syntactic edges corresponding to an arc.

Parameters: *1.* `option` The option of interest *2.* `accumulator` A function taking two values. When there are more than one syntactic edges corresponding to `self` for which the `option` is set, this function will be called repeatedly for the different values. The first time it will be called for the first two values. Next, it will be called for the result of this call and the third value, and so on. *3.* `only_aligned` A boolean. If true, only the aligned syntactic edges will be considered.

Returns: *1.* If the option is not set for any (aligned) syntactic edges corresponding to `self`, `nil` is returned. If there is exactly one edge, the value of this edge is returned. Otherwise, the result of repeatedly applying the `accumulator` function as described above.

The result is cached, repeated calls will not invoke the `accumulator` function again.

Usage: Here is typical usage:

```
local total_length = a:optionsAccumulated('length', function (a,b) return a+b end) or 0
```

`function Arc:syntacticTailAndHead()`

Compute the syntactic head and tail of an arc. For this, we have a look at the syntactic digraph underlying the arc. If there is at least once syntactic edge going from the arc's tail to the arc's head, the arc's tail and head are returned. Otherwise, we test whether there is a syntactic edge in the other direction and, if so, return head and tail in reverse order. Finally, if there is no syntactic edge at all corresponding to the arc in either direction, `nil` is returned.

Returns: *1.* The syntactic tail

Returns: *1.* The syntactic head

`function Arc:pointCloud()`

Compute the point cloud.

Returns: *1.* This method will return the "point cloud" of an arc, which is an array of all points that must be rotated and shifted along with the endpoints of an edge.

`function Arc:eventIndex()`

Compute an event index for the arc.

Returns: *1.* The lowest event index of any edge involved in the arc (or nil, if there is no syntactic edge).

`function Arc:spanPriority()`

The span collector

This method returns the top (that is, smallest) priority of any edge involved in the arc.

The priority of an edge is computed as follows:

1. If the option `"span priority"` is set, this number will be used.
2. If the edge has the same head as the arc, we lookup the key `"span priority " .. edge.direction`. If set, we use this value.
3. If the edge has a different head from the arc (the arc is "reversed" with respect to the syntactic edge), we lookup the key `"span priority reversed " .. edge.direction`. If set, we use this value.
4. Otherwise, we use priority 5.

Returns: 1. The priority of the arc, as described above.

`function Arc:sync()`

Sync an `Arc` with its syntactic edges with respect to the path and generated options. It causes the following to happen: If the `path` field of the arc is `nil`, nothing happens with respect to the path. Otherwise, a copy of the `path` is created. However, for every path element that is a function, this function is invoked with the syntactic edge as its parameter. The result of this call should now be a `Coordinate`, which will replace the function in the `Path`.

You use this method like this:

```
...
local arc = g:connect(s,t)
arc:setPolylinePath { Coordinate.new(x,y), Coordinate.new(x1,y1) }
...
arc:sync()
```

Next, similar to the path, the field `generated_options` is considered. If it is not `nil`, then all options listed in this field are appended to all syntactic edges underlying the arc.

Note that this function will automatically be called for all arcs of the `ugraph`, the `digraph`, and the `spanning_tree` of an algorithm by the rendering pipeline.

`function Arc:tailAnchorForArcPath()`

This method returns a "coordinate factory" that can be used as the coordinate of a `moveto` at the beginning of a path starting at the `tail` of the arc. Suppose you want to create a path starting at the tail vertex, going to the coordinate $(10, 10)$ and ending at the head vertex. The trouble is that when you create the path corresponding to this route, you typically do not know where the tail vertex is going to be. Even if that *has* already been settled, you will still have the problem that different edges underlying the arc may wish to start their paths at different anchors inside the tail vertex. In such cases, you use this method to get a function that will, later on, compute the correct position of the anchor as needed.

Here is the code you would use to create the above-mentioned path:

```
local a = g:connect(tail,head)
...
arc.path = Path.new()
arc.path:appendMoveto(arc:tailAnchorForArcPath())
arc.path:appendLineto(10, 10)
arc.path:appendLineto(arc:headAnchorForArcPath())
```

Normally, however, you will not write code as detailed as the above and you would just write instead of the last three lines:

```
arc:setPolylinePath { Coordinate.new (10, 10) }
```

`function Arc:headAnchorForArcPath()`

See `Arc:tailAnchorForArcPath`.

`function Arc:setPolylinePath(coordinates)`

Setup the `path` field of an arc in such a way that it corresponds to a sequence of straight line segments starting at the tail's anchor and ending at the head's anchor.

Parameters: 1. `coordinates` An array of `Coordinates` through which the line will go through.

36.5.4 Edges

Lua table `Edge` (declared in `pgf.gd.model.Edge`)

An `Edge` is a "syntactic" connection between two vertices that represents a connection present in the syntactic digraph. Unlike an `Arc`, `Edge` objects are not controlled by the `Digraph` class. Also unlike `Arc` objects, there can be several edges between the same vertices, namely whenever several such edges are present in the syntactic digraph.

In detail, the relationship between arcs and edges is as follows: If there is an `Edge` between two vertices u and v in the syntactic digraph, there will be an `Arc` from u to v and the array `syntactic_edges` of this `Arc` object will contain the `Edge` object. In particular, if there are several edges between the same vertices, all of these edges will be part of the array in a single `Arc` object.

Edges, like arcs, are always directed from a `tail` vertex to a `head` vertex; this is true even for undirected vertices. The `tail` vertex will always be the vertex that came first in the syntactic specification of the edge, the `head` vertex is the second one. Whether an edge is directed or not depends on the `direction` of the edge, which may be one of the following:

1. `"->"`
2. `"--"`
3. `"<-"`
4. `"<->"`
5. `"-!-"`

Field `head`

The head vertex of this edge.

Field `tail`

The tail vertex of this edge.

Field `event`

The creation `Event` of this edge.

Field `options`

A table of options that contains user-defined options.

Field `direction`

One of the directions named above.

Field `path`

A `Path` object that describes the path of the edge. The path's coordinates are interpreted *absolutely*.

Field `generated_options`

This is an options array that is generated by the algorithm. When the edge is rendered later on, this array will be passed back to the display layer. The syntax is the same as for the `declare_parameter_sequence` function, see `InterfaceToAlgorithms`.

Field `animations`

An array of animations, see the `animations` field of the `Vertex` class for the syntax.

Alphabetical method summary:

`function Edge.new (values)`

`function Edge:headAnchorForEdgePath ()`

```
function Edge:setPolylinePath (coordinates)
function Edge:tailAnchorForEdgePath ()
```

`function Edge.new(values)`

Create a new edge. The `initial` parameter allows you to setup some initial values.

Usage:

```
local v = Edge.new { tail = v1, head = v2 }
```

Parameters: 1. `initial` Values to override defaults. –

Returns: 1. A new edge object.

`function Edge:tailAnchorForEdgePath()`

This method returns a "coordinate factory" that can be used as the coordinate of a `moveto` at the beginning of a path starting at the `tail` of the arc. Suppose you want to create a path starting at the tail vertex, going to the coordinate $(10, 10)$ and ending at the head vertex. The trouble is that when you create the path corresponding to this route, you typically do not know where the tail vertex is going to be. In this case, you use this method to get a function that will, later on, compute the correct position of the anchor as needed.

Note that you typically do not use this function, but use the corresponding function of the `Arc` class. Use this function only if there are multiple edges between two vertices that need to be routed differently.

Here is the code you would use to create the above-mentioned path:

```
local a = g:connect(tail,head)
local e = a.syntactic_edges[1]
...
e.path = Path.new()
e.path:appendMoveto(e:tailAnchorForEdgePath())
e.path:appendLineto(10, 10)
e.path:appendLineto(e:headAnchorForEdgePath())
```

As for the `Arc` class, you can also setup a polyline more easily:

```
e:setPolylinePath { Coordinate.new (10, 10) }
```

`function Edge:headAnchorForEdgePath()`

See `Arc:tailAnchorForArcPath`.

`function Edge:setPolylinePath(coordinates)`

Setup the `path` field of an edge in such a way that it corresponds to a sequence of straight line segments starting at the tail's anchor and ending at the head's anchor.

Parameters: 1. `coordinates` An array of `Coordinates` through which the line will go through.

36.5.5 Collections

Lua table `Collection` (declared in `pgf.gd.model.Collection`)

A collection is essentially a subgraph of a graph, that is, a "collection" of some nodes and some edges of the graph. The name "collection" was chosen over "subgraph" since the latter are often thought of as parts of a graph that are rendered in a special way (such as being surrounded by a rectangle), while collections are used to model such diverse things as hyperedges, sets of vertices that should be on the same level in a layered algorithm, or, indeed, subgraphs that are rendered in a special way.

Collections are grouped into "kinds". All collections of a given kind can be accessed by algorithms through an array whose elements are the collections. On the display layer, for each kind a separate key is available to indicate that a node or an edge belongs to a collection.

Collections serve two purposes: First, they can be seen as "hints" to graph drawing algorithms that certain nodes and/or edges "belong together". For instance, collections of kind `same layer` are used by the Sugiyama algorithm to group together nodes that should appear at the same height of the output. Second, since collections are also passed back to the display layer in a postprocessing step, they can be used to render complicated concepts such as hyperedges (which are just collections of nodes, after all) or subgraphs.

Field `kind`

The "kind" of the collection.

Field `vertices`

A lookup table of vertices (that is, both an array with the vertices in the order in which they appear as well as a table such that `vertices[vertex] == true` whenever `vertex` is present in the table.

Field `edges`

A lookup table of edges (not arcs!).

Field `options`

An options table. This is the table of options that was in force when the collection was created.

Field `child_collections`

An array of all collections that are direct children of this collection (that is, they were defined while the current collection was the most recently defined collection on the options stack). However, you should use the methods `children`, `descendants`, and so to access this field.

Field `parent_collection`

The parent collection of the current collection. This field may be `nil` in case a collection has no parent.

Field `event`

An `Event` object that was create for this collection. Its `kind` will be `"collection"` while its `parameter` will be the collection kind.

Alphabetical method summary:

```
function Collection.new (t)
function Collection:childrenOfKind (kind)
function Collection:children ()
function Collection:descendantsOfKind (kind)
function Collection:descendants ()
```

`function Collection.new(t)`

Creates a new collection. You should not call this function directly, it is called by the interface classes.

Parameters: 1. `t` A table of initial values. The field `t.kind` must be a nonempty string.

Returns: 1. The new collection

`function Collection:children()`

A collection can have any number of *child collections*, which are collections nested inside the collection. You can access the array of these children through this method. You may not modify the array returned by this function.

Returns: 1. The array of children of `self`.

`function Collection:childrenOfKind(kind)`

This method works like the `children` method. However, the tree of collections is, conceptually, contracted by considering only these collections that have the `kind` given as parameter. For instance, if `self` has a child collection of a kind different from `kind`, but this child collection has, in turn, a child collection of kind `kind`, this latter child collection will be included in the array – but not any of its child collections.

Parameters: 1. `kind` The collection kind to which the tree of collections should be restricted.

Returns: 1. The array of children of `self` in this contracted tree.

`function Collection:descendants()`

The descendants of a collection are its children, plus their children, plus their children, and so on.

Returns: 1. An array of all descendants of `self`. It will be in preorder.

`function Collection:descendantsOfKind(kind)`

The descendants of a collection of the given `kind`.

Parameters: *1.* `kind` A collection kind.

Returns: *1.* An array of all descendants of `self` of the given `kind`.

36.5.6 Coordinates, Paths, and Transformations

Lua table `Coordinate` (declared in `pgf.gd.model.Coordinate`)

A Coordinate models a position on the drawing canvas.

It has an `x` field and a `y` field, which are numbers that will be interpreted as TEX points (1/72.27th of an inch). The x-axis goes right and the y-axis goes up.

Field `x`

Field `y`

There is also a static field called `origin` that is always equal to the origin.

Alphabetical method summary:

```
function Coordinate.__add (a,b)
function Coordinate.__div (a,b)
function Coordinate.__mul (a,b)
function Coordinate.__sub (a,b)
function Coordinate.__unm (a)
function Coordinate.boundingBox (array)
function Coordinate.new (x,y)
function Coordinate:apply (t)
function Coordinate:clone ()
function Coordinate:moveTowards (c,f)
function Coordinate:normalized ()
function Coordinate:normalize ()
function Coordinate:norm ()
function Coordinate:scale (s)
function Coordinate:shiftByCoordinate (c)
function Coordinate:shift (a,b)
function Coordinate:unshiftByCoordinate (c)
function Coordinate:unshift (a,b)
```

`function Coordinate.new(x,y)`

Creates a new coordinate.

Parameters: *1.* `x` The x value *2.* `y` The y value

Returns: *1.* A coordinate

`function Coordinate:clone()`

Creates a new coordinate that is a copy of an existing one.

Returns: *1.* A new coordinate at the same location as `self`

`function Coordinate:apply(t)`

Apply a transformation matrix to a coordinate, see `pgf.gd.lib.Transform` for details.

Parameters: *1.* `t` A transformation.

`function Coordinate:shift(a,b)`

Shift a coordinate

Parameters: *1.* `a` An x offset *2.* `b` A y offset

`function Coordinate:unshift(a,b)`

"Unshift" a coordinate (which is the same as shifting by the inversed coordinate; only faster).

Parameters: *1.* `a` An x offset *2.* `b` A y offset

`function Coordinate:shiftByCoordinate(c)`

Like `shift`, only for coordinate parameters.

Parameters: *1.* `c` Another coordinate. The x- and y-values of `self` are increased by the x- and y-values of this coordinate.

`function Coordinate:unshiftByCoordinate(c)`

Like `unshift`, only for coordinate parameters.

Parameters: *1.* `c` Another coordinate.

`function Coordinate:moveTowards(c,f)`

Moves the coordinate a fraction of `f` along a straight line to `c`.

Parameters: *1.* `c` Another coordinate *2.* `f` A fraction

`function Coordinate:scale(s)`

Scale a coordinate by a factor

Parameters: *1.* `s` A factor.

`function Coordinate.__add(a,b)`

Add two coordinates, yielding a new coordinate. Note that it will be a lot faster to call shift, whenever this is possible.

Parameters: *1.* `a` A coordinate *2.* `b` A coordinate

`function Coordinate.__sub(a,b)`

Subtract two coordinates, yielding a new coordinate. Note that it will be a lot faster to call unshift, whenever this is possible.

Parameters: *1.* `a` A coordinate *2.* `b` A coordinate

`function Coordinate.__unm(a)`

The unary minus (mirror the coordinate against the origin).

Parameters: *1.* `a` A coordinate

`function Coordinate.__mul(a,b)`

The multiplication operator. Its effect depends on the parameters: If both are coordinates, their dot-product is returned. If exactly one of them is a coordinate and the other is a number, the scalar multiple of this coordinate is returned.

Parameters: *1.* `a` A coordinate or a scalar *2.* `b` A coordinate or a scalar

Returns: *1.* The dot product or scalar product.

`function Coordinate.__div(a,b)`

The division operator. Returns the scalar division of a coordinate by a scalar.

Parameters: *1.* `a` A coordinate *2.* `b` A scalar (not equal to zero).

Returns: *1.* The scalar product or a * (1/b).

`function Coordinate:norm()`

The norm function. Returns the norm of a coordinate.

Parameters: *1.* `a` A coordinate

Returns: *1.* The norm of the coordinate

`function Coordinate:normalize()`

Normalize a vector: Ensure that it has length 1. If the vector used to be the 0-vector, it gets replaced by (1,0).

`function Coordinate:normalized()`

Normalized version of a vector: Like `normalize`, only the result is returned in a new vector.

Returns: 1. Normalized version of `self`

`function Coordinate.boundingBox(array)`

Compute a bounding box around an array of coordinates

Parameters: 1. `array` An array of coordinates

Returns: 1. `min_x` The minimum x value of the bounding box of the array

Returns: 1. `min_y` The minimum y value

Returns: 1. `max_x`

Returns: 1. `max_y`

Returns: 1. `center_x` The center of the bounding box

Returns: 1. `center_y`

Lua table `Path` (declared in `pgf.gd.model.Path`)

A Path models a path in the plane.

Following the PostScript/PDF/SVG convention, a path consists of a series of path segments, each of which can be closed or not. Each path segment, in turn, consists of a series of Bézier curves and straight line segments; see Section 14 for an introduction to paths in general.

A `Path` object is a table whose array part stores `Coordinate` objects, `strings`, and `functions` that describe the path of the edge. The following strings are allowed in this array:

- `"moveto"` The line's path should stop at the current position and then start anew at the next coordinate in the array.
- `"lineto"` The line should continue from the current position to the next coordinate in the array.
- `"curveto"` The line should continue form the current position with a Bézier curve that is specified by the next three `Coordinate` objects (in the usual manner).
- `"closepath"` The line's path should be "closed" in the sense that the current subpath that was started with the most recent moveto operation should now form a closed curve.

Instead of a `Coordinate`, a `Path` may also contain a function. In this case, the function, when called, must return the `Coordinate` that is "meant" by the position. This allows algorithms to add coordinates to a path that are still not fixed at the moment they are added to the path.

Alphabetical method summary:

`function Path.new (initial)`

`function Path:appendArcTo (target,radius_or_center,clockwise,trans)`

`function Path:appendArc (start_angle,end_angle,radius,trans)`

`function Path:appendClosepath ()`

`function Path:appendCurveto (a,b,c,d,e,f)`

`function Path:appendLineto (x,y)`

`function Path:appendMoveto (x,y)`

`function Path:boundingBox ()`

`function Path:clear ()`

`function Path:clone ()`

`function Path:coordinates ()`

`function Path:cutAtBeginning (index, time)`

`function Path:cutAtEnd (index, time)`

`function Path:intersectionsWith (path)`

`function Path:makeRigid ()`

`function Path:pad (padding)`

`function Path:reversed ()`

`function Path:shiftByCoordinate (x)`

`function Path:shift (x,y)`

`function Path:transform (t)`

`function Path.new(initial)`

Creates an empty path.

Parameters: *1.* `initial` A table containing an array of strings and coordinates that constitute the path. Coordinates may be given as tables or as a pair of numbers. In this case, each pair of numbers is converted into one coordinate. If omitted, a new empty path is created.

Returns: *1.* A empty Path

`function Path:clone()`

Creates a copy of a path.

Returns: *1.* A copy of the path

`function Path:reversed()`

Returns the path in reverse order.

Returns: *1.* A copy of the reversed path

`function Path:transform(t)`

Transform all points on a path.

Parameters: *1.* `t` A transformation, see `pgf.gd.lib.Transform`. It is applied to all `Coordinate` objects on the path.

`function Path:shift(x,y)`

Shift all points on a path.

Parameters: *1.* `x` An x-shift *2.* `y` A y-shift

`function Path:shiftByCoordinate(x)`

Shift by all points on a path.

Parameters: *1.* `x` A coordinate

`function Path:clear()`

Makes the path empty.

`function Path:appendMoveto(x,y)`

Appends a `moveto` to the path.

Parameters: *1.* `x` A `Coordinate` or `function` or, if the `y` parameter is not `nil`, a number that is the x-part of a coordinate. *2.* `y` The y-part of the coordinate.

`function Path:appendLineto(x,y)`

Appends a `lineto` to the path.

Parameters: *1.* `x` A `Coordinate` or `function`, if the `y` parameter is not `nil`, a number that is the x-part of a coordinate. *2.* `y` The y-part of the coordinate.

`function Path:appendClosepath()`

Appends a `closepath` to the path.

`function Path:appendCurveto(a,b,c,d,e,f)`

Appends a `curveto` to the path. There can be either three coordinates (or functions) as parameters (the two support points and the target) or six numbers, where two consecutive numbers form a `Coordinate`. Which case is meant is detected by the presence of a sixth non-nil parameter.

`function Path:makeRigid()`

Makes a path "rigid", meaning that all coordinates that are only given as functions are replaced by the values these functions yield.

`function Path:coordinates()`

Returns an array of all coordinates that are present in a path. This means, essentially, that all strings are filtered out.

Returns: *1.* An array of all coordinate objects on the path.

`function Path:boundingBox()`

Returns a bounding box of the path. This will not necessarily be the minimal bounding box in case the path contains curves because, then, the support points of the curve are used for the computation rather than the actual bounding box of the path.

If the path contains no coordinates, all return values are 0.

Returns: *1.* `min_x` The minimum x value of the bounding box of the path

Returns: *1.* `min_y` The minimum y value

Returns: *1.* `max_x`

Returns: *1.* `max_y`

Returns: *1.* `center_x` The center of the bounding box

Returns: *1.* `center_y`

`function Path:intersectionsWith(path)`

Computes all intersections of a path with another path and returns them as an array of coordinates. The intersections will be sorted "along the path `self`". The implementation uses a divide-and-conquer approach that should be reasonably fast in practice.

Parameters: *1.* `path` Another path

Returns: *1.* Array of all intersections of `path` with `self` in the order they appear on `self`. Each entry of this array is a table with the following fields:

- `index` The index of the segment in `self` where the intersection occurs.
- `time` The "time" at which a point traveling along the segment from its start point to its end point.
- `point` The point itself.

`function Path:cutAtBeginning(index, time)`

Shorten a path at the beginning. We are given the index of a segment inside the path as well as a point in time along this segment. The path is now shortened so that everything before this segment and everything in the segment before the given time is removed from the path.

Parameters: *1.* `index` The index of a path segment. *2.* `time` A time along the specified path segment.

`function Path:cutAtEnd(index, time)`

Shorten a path at the end. This method works like `cutAtBeginning`, only the path is cut at the end.

Parameters: *1.* `index` The index of a path segment. *2.* `time` A time along the specified path segment.

`function Path:pad(padding)`

"Pads" the path. The idea is the following: Suppose we stroke the path with a pen whose width is twice the value `padding`. The outer edge of this stroked drawing is now a path by itself. The path will be a bit longer and "larger". The present function tries to compute an approximation to this resulting path.

The algorithm used to compute the enlarged part does not necessarily compute the precise new path. It should work correctly for polyline paths, but not for curved paths.

Parameters: *1.* `padding` A padding distance.

Returns: *1.* The padded path.

`function Path:appendArc(start_angle,end_angle,radius,trans)`

Appends an arc (as in the sense of "a part of the circumference of a circle") to the path. You may optionally provide a transformation matrix, which will be applied to the arc. In detail, the following happens: We first invert the transformation and apply it to the start point. Then we compute the arc "normally", as if no transformation matrix were present. Then we apply the transformation matrix to all computed points.

Parameters: *1.* `start_angle` The start angle of the arc. Must be specified in degrees. *2.* `end_angle` the end angle of the arc. *3.* `radius` The radius of the circle on which this arc lies. *4.* `trans` A transformation matrix. If `nil`, the identity matrix will be assumed.

`function Path:appendArcTo(target,radius_or_center,clockwise,trans)`

Appends a clockwise arc (as in the sense of "a part of the circumference of a circle") to the path such that it ends at a given point. If a transformation matrix is given, both start and end point are first transformed according to the inverted transformation, then the arc is computed and then transformed back.

Parameters: *1.* `target` The point where the arc should end. *2.* `radius_or_center` If a number, it is the radius of the circle on which this arc lies. If it is a `Coordinate`, this is the center of the circle. *3.* `clockwise` If true, the arc will be clockwise. Otherwise (the default, if nothing or `nil` is given), the arc will be counter clockwise. *4.* `trans` A transformation matrix. If missing, the identity matrix is assumed.

Lua table `Transform` (declared in `pgf.gd.lib.Transform`)

The `Transform` table provides a set of static methods for creating and handling canvas transformation matrices. Such a matrix is actually just an array of six numbers. The idea is that "applying" an array a, b, c, d, e, f a vector (x, y) will yield the new vector $(ax + by + e, cx + dy + f)$. For details on how such matrices work, see Section 108.2.1

Alphabetical method summary:

`function Transform.concat (a,b)`

`function Transform.invert (t)`

`function Transform.new_rotation (angle)`

`function Transform.new_scaling (x_scale, y_scale)`

`function Transform.new_shift (x,y)`

`function Transform.new (a,b,c,d,x,y)`

`function Transform.new(a,b,c,d,x,y)`

Creates a new transformation array.

Parameters: *1.* `a` First component *2.* `b` Second component *3.* `c` Third component *4.* `d` Fourth component *5.* `x` The x shift *6.* `y` The y shift

Returns: *1.* A transformation object.

`function Transform.new_shift(x,y)`

Creates a new transformation object that represents a shift.

Parameters: *1.* `x` An x-shift *2.* `y` A y-shift

Returns: *1.* A transformation object

`function Transform.new_rotation(angle)`

Creates a new transformation object that represents a rotation.

Parameters: *1.* `angle` An angle

Returns: *1.* A transformation object

`function Transform.new_scaling(x_scale, y_scale)`

Creates a new transformation object that represents a scaling.

Parameters: *1.* `x` The horizontal scaling *2.* `y` The vertical scaling (if missing, the horizontal scaling is used)

Returns: *1.* A transformation object

`function Transform.concat(a,b)`

Concatenate two transformation matrices, returning the new one.

Parameters: *1.* `a` The first transformation *2.* `b` The second transformation

Returns: *1.* The transformation representing first applying `b` and then applying `a`.

`function Transform.invert(t)`

Inverts a transformation matrix.

Parameters: 1. `t` The transformation.

Returns: 1. The inverted transformation

36.5.7 Options and Data Storages for Vertices, Arcs, and Digraphs

Many objects in the graph drawing system have an `options` table attached to them. These tables will contain the different kinds options specified by the user for the object. For efficiency reasons, many objects may share the same options table (since, more often than not, almost all objects have exactly the same `options` table). For this reason, you cannot store anything in an options table, indeed, you should never attempt to write anything into an options table. Instead, you should use a `Storage`.

Lua table `Storage` (declared in `pgf.gd.lib.Storage`)

A storage is an object that, as the name suggests, allows you to "store stuff concerning objects." Basically, it behaves like table having weak keys, which means that once the objects for which you "store stuff" go out of scope, they are also removed from the storage. Also, you can specify that for each object of the storage you store a table. In this case, there is no need to initialize this table for each object; rather, when you write into such a table and it does not yet exist, it is created "on the fly".

The typical way you use storages is best explained with the following example: Suppose you want to write a depth-first search algorithm for a graph. This algorithm might wish to mark all nodes it has visited. It could just say `v.marked = true`, but this might clash with someone else also using the `marked` key. The solution is to create a `marked` storage. The algorithm can first say

```
local marked = Storage.new()
```

and then say

```
marked[v] = true
```

to mark its objects. The `marked` storage object does not need to be created locally inside a function, you can declare it as a local variable of the whole file; nevertheless, the entries for vertices no longer in use get removed automatically. You can also make it a member variable of the algorithm class, which allows you make the information about which objects are marked globally accessible.

Now suppose the algorithm would like to store even more stuff in the storage. For this, we might use a table and can use the fact that a storage will automatically create a table when necessary:

```
local info = Storage.newTableStorage()

info[v].marked = true  -- the "info[v]" table is
                       -- created automatically here

info[v].foo    = "bar"
```

Again, once `v` goes out of scope, both it and the info table will removed.

Alphabetical method summary:

`function Storage.newTableStorage ()`

`function Storage.new ()`

`function Storage.new()`

Create a new storage object.

Returns: 1. A new `Storage` instance.

`function Storage.newTableStorage()`

Create a new storage object which will install a table for every entry automatically.

Returns: 1. A new `Storage` instance.

36.5.8 Events

Lua table `Event` (declared in `pgf.gd.lib.Event`)

Events are used to communicate "interesting" events from the parser to the graph drawing algorithms.

As a syntactic description of some graph is being parsed, vertices, arcs, and a digraph object representing this graph get constructed. However, even though syntactic annotations such as options for the vertices and arcs are attached to them and can be accessed through the graph objects, some syntactic information is neither represented in the digraph object nor in the vertices and the arcs. A typical example is a "missing" node in a tree: Since it is missing, there is neither a vertex object nor arc objects representing it. It is also not a global option of the graph.

For these reasons, in addition to the digraph object itself, additional information can be passed by a parser to graph drawing algorithms through the means of events. Each `Event` consists of a `kind` field, which is just some string, and a `parameters` field, which stores additional, kind-specific information. As a graph is being parsed, a string of events is accumulated and is later on available through the `events` field of the graph drawing scope.

The following events are created during the parsing process by the standard parsers of Ti*k*Z:

`node` When a node of the input graph has been parsed and a `Vertex` object has been created for it, an event with kind `node` is created. The `parameter` of this event is the just-created vertex.

The same kind of event is used to indicate "missing" nodes. In this case, the `parameters` field is `nil`.

`edge` When an edge of the input graph has been parsed, an event is created of kind `edge`. The `parameters` field will store an array with two entries: The first is the `Arc` object whose `syntactic_edges` field stores the `edge`. The second is the index of the edge inside the `syntactic_edges` field.

`begin` Signals the beginning of a group, which will be ended with a corresponding `end` event later on. The `parameters` field will indicate the kind of group. Currently, only the string `"descendants"` is used as `parameters`, indicating the start of several nodes that are descendants of a given node. This information can be used by algorithms for reconstructing the input structure of trees.

`end` Signals the end of a group begun by a `begin` event earlier on.

Field `kind`

A string representing the kind of the events.

Field `parameters`

Kind-specific parameters.

Field `index`

A number that stores the events logical position in the sequence of events. The number need not be an integer array index.

Alphabetical method summary:

`function Event.new (values)`

`function Event.new(values)`

Create a new event object

Parameters: 1. `initial` Initial fields of the new event.

Returns: 1. The new object

36.6 Graph Transformations

36.6.1 The Layout Pipeline

Lua table `LayoutPipeline` (declared in `pgf.gd.control.LayoutPipeline`)

This class controls the running of graph drawing algorithms on graphs. In particular, it performs pre- and posttransformations and also invokes the collapsing of sublayouts.

You do not call any of the methods of this class directly, the whole class is included only for documentation purposes.

Before an algorithm is applied, a number of transformations will have been applied, depending on the algorithm's `preconditions` field:

- `connected`
  If this property is set for an algorithm (that is, in the `declare` statement for the algorithm the `predconditions` field has the entry `connected=true` set), then the graph will be decomposed into connected components. The algorithm is run on each component individually.
- `tree`
  When set, the field `spanning_tree` of the algorithm will be set to a spanning tree of the graph. This option implies `connected`.
- `loop_free`
  When set, all loops (arcs from a vertex to itself) will have been removed when the algorithm runs.
- `at_least_two_nodes`
  When explicitly set to `false` (this precondition is `true` by default), the algorithm will even be run if there is only a single vertex in the graph.

Once the algorithm has run, the algorithm's `postconditions` will be processed:

- `upward_oriented`
  When set, the algorithm tells the layout pipeline that the graph has been laid out in a layered manner with each layer going from left to right and layers at a whole going upwards (positive y-coordinates). The graph will then be rotated and possibly swapped in accordance with the `grow` key set by the user.
- `fixed`
  When set, no rotational postprocessing will be done after the algorithm has run. Usually, a graph is rotated to meet a user's `orient` settings. However, when the algorithm has already "ideally" rotated the graph, set this postcondition.

In addition to the above-described always-present and automatic transformations, users may also specify additional pre- and posttransformations. This happens when users install additional algorithms in appropriate phases. In detail, the following happens in order:

1. If specified, the graph is decomposed into connected components and the following steps are applied to each component individually.
2. All algorithms in the phase stack for the phase `preprocessing` are applied to the component. These algorithms are run one after the other in the order they appear in the phase stack.
3. If necessary, the spanning tree is now computed and rotational information is gathered.
4. The single algorithm in phase `main` is called.
5. All algorithms in the phase stack for the phase `edge routing` are run.
6. All algorithms in the phase stack for phase `postprocessing` are run.
7. Edge syncing, orientation, and anchoring are applied.

If sublayouts are used, all of the above (except for anchoring) happens for each sublayout.

Alphabetical method summary:

```
function LayoutPipeline.anchor (graph, scope)
function LayoutPipeline.cutEdges (graph)
function LayoutPipeline.decompose (digraph)
function LayoutPipeline.orient (rotation_info, postconditions, graph, scope)
function LayoutPipeline.packComponents (syntactic_digraph, components)
function LayoutPipeline.prepareBoundingBoxes (rotation_info, adjusted_bb, graph, vertices)
function LayoutPipeline.prepareRotateAround (postconditions, graph)
```

`function LayoutPipeline.anchor(graph, scope)`

This function is called internally to perform the graph anchoring procedure described in Section 28.4. These transformations are always performed.

Parameters: *1.* `graph` A graph *2.* `scope` The scope

`function LayoutPipeline.prepareRotateAround(postconditions, graph)`

This method tries to determine in which direction the graph is supposed to grow and in which direction the algorithm will grow the graph. These two pieces of information together produce a necessary rotation around some node. This rotation is returned in a table.

Note that this method does not actually cause a rotation to happen; this is left to other method.

Parameters: *1.* `postconditions` The algorithm's postconditions. *2.* `graph` An undirected graph

Returns: *1.* A table containing the computed information.

`function LayoutPipeline.prepareBoundingBoxes(rotation_info, adjusted_bb, graph, vertices)`

Compute growth-adjusted node sizes.

For each node of the graph, compute bounding box of the node that results when the node is rotated so that it is in the correct orientation for what the algorithm assumes.

The "bounding box" actually consists of the fields

- `sibling_pre`,
- `sibling_post`,
- `layer_pre`, and
- `layer_post`,

which correspond to "min x", "min y", "min y", and "max y" for a tree growing up.

The computation of the "bounding box" treats a centered circle in a special way, all other shapes are currently treated like a rectangle.

Parameters: *1.* `rotation_info` The table computed by the function prepareRotateAround *2.* `packing_storage` A storage in which the computed distances are stored. *3.* `graph` An graph *4.* `vertices` An array of to-be-prepared vertices inside graph

`function LayoutPipeline.orient(rotation_info, postconditions, graph, scope)`

Performs a post-layout orientation of the graph by performing the steps documented in Section 28.5.

Parameters: *1.* `rotation_info` The info record computed by the function `prepareRotateAround`. *2.* `postconditions` The algorithm's postconditions. *3.* `graph` A to-be-oriented graph. *4.* `scope` The graph drawing scope.

`function LayoutPipeline.decompose(digraph)`

This internal function is called to decompose a graph into its components. Whether or not this function is called depends on whether the precondition `connected` is set for the algorithm class and whether the `componentwise` key is used.

Parameters: *1.* `graph` A to-be-decomposed graph

Returns: *1.* An array of graph objects that represent the connected components of the graph.

`function LayoutPipeline.packComponents(syntactic_digraph, components)`

This internal function packs the components of a graph. See Section 28.7 for details.

Parameters: *1.* `graph` The graph *2.* `components` A list of components

`function LayoutPipeline.cutEdges(graph)`

Cut the edges. This function handles the "cutting" of edges. The idea is that every edge is a path going from the center of the from node to the center of the target node. Now, we intersect this path with the path of the start node and cut away everything before this intersection. Likewise, we intersect the path with the head node and, again, cut away everything following the intersection.

These cuttings are not done if appropriate options are set.

36.6.2 Hints For Edge Routing

Lua table `Hints` (declared in `pgf.gd.routing.Hints`)

The `Hints` class provides a way for graph drawing algorithms to communicate certain possibilities concerning the routing of edges to edge routing algorithms. This partly decouples the choice of the vertex positioning algorithms from the choice of edge routing algorithm. For instance, for a simple necklace routing, it is unclear whether the edges on the necklace should be routing "along the necklace" or not. Thus, necklace routing algorithms will "hint" that a necklace is present and only when the `necklace routing` algorithm is selected will these hints lead to actual bending of edges.

For each kind of hint, there are methods in this class for creating the hints and other methods for reading them. Hints are always local to the ugraph.

Alphabetical method summary:

```
function Hints.addNecklaceCircleHint (ugraph, necklace, center, clockwise)
function Hints.getNecklaceHints (ugraph)
```

`function Hints.addNecklaceCircleHint(ugraph, necklace, center, clockwise)`

Adds a necklace hint. In this case, the hint indicates that the given sequence of vertices lie on a circle.

The idea is that an algorithm may specify that in a given graph certain sequences of nodes form a "necklace", which is typically a circle. There may be more than one necklace inside a given graph. For each necklace, whenever an arc connects subsequent nodes on the necklace, they get bend in such a way that they lie follow the path of the necklace. If an arc lies on more than one necklace, the "last one wins".

Parameters: 1. `ugraph` The ugraph to which this hint is added *2.* `necklace` The sequence of vertices that form the necklace. If the necklace is closed, the last vertex must equal the first one. *3.* `center` If provided, must be `Coordinate` that specifies the center of the circle on which the vertices lie. If not provided, the origin is assumed. *4.* `clockwise` If `true`, the vertices are in clockwise order, otherwise in counter-clockwise order.

`function Hints.getNecklaceHints(ugraph)`

Gets the necklace hints.

This function will return an array whose entries are necklace hints. Each entry in the array has a `necklace` field, which is the field passed to the `addNecklaceXxxx` methods. For a circle necklace, the `center` and `clockwise` fields will be set. (Other necklaces are not yet implemented.)

Parameters: 1. `ugraph` The ugraph for which the necklace hints are requested.

Returns: 1. The array of necklaces as described above.

36.7 The Interface To Algorithms

Lua table `InterfaceToAlgorithms` (declared in `pgf.gd.interface.InterfaceToAlgorithms`)

This class provides the interface between the graph drawing system and algorithms. Another class, `InterfaceToDisplay`, binds the display layers (like Ti*k*Z or a graph drawing editor) to the graph drawing system "from the other side".

The functions declared here can be used by algorithms to communicate with the graph drawing system, which will usually forward the "requests" of the algorithms to the display layers in some way. For instance, when you declare a new parameter, this parameter will become available on the display layer.

Alphabetical method summary:

```
function InterfaceToAlgorithms.addHandler (test, handler)
function InterfaceToAlgorithms.createEdge (algorithm, tail, head, init)
function InterfaceToAlgorithms.createVertex (algorithm, init)
function InterfaceToAlgorithms.declare (t)
function InterfaceToAlgorithms.findVertexByName (name)
```

`function InterfaceToAlgorithms.addHandler(test, handler)`

Adds a handler for the `declare` function. The `declare` command is just a "dispatcher" to one of many possible declaration functions. Which function is used, depends on which fields are present in the table

passed to `declare`. For each registered handler, we call the `test` function. If it returns neither `nil` nor `false`, the `handler` field of this handler is called. If it returns `true`, the handler immediately finishes. Otherwise, the next handler is tried.

`function InterfaceToAlgorithms.declare(t)`

This function is the "work-horse" for declaring things. It allows you to specify on the algorithmic layer that a key "is available" for use on the display layer. There is just one function for handling all declarations in order to make the declarations easy-to-use since you just need to import a single function:

```
local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare
```

You can now use `declare` it as follows: You pass it a table containing information about the to-be-declared key. The table *must* have a field `key` whose value is unique and must be a string. If the value of `key` is, say, `"foo"`, the parameter can be set on the display layer such as, say, the Ti*k*Z layer, using `/graph drawing/foo`. Here is a typical example of how a declaration is done:

```
---
declare {
  key     = "electrical charge",
  type    = "number",
  initial = "1.0",

  summary = "The ``electrical charge'' is a property...",
  documentation = [[...]],
  examples = [[...]]
}
```

Inlining Documentation. The three keys `summary`, `documentation` and `examples` are intended for the display layer to give the users information about what the key does. The `summary` should be a string that succinctly describes the option. This text will typically be displayed for instance as a "tool tip" or in an option overview. The `documentation` optionally provides more information and should be typeset using TEX. The `examples` can either be a single string or an array of strings. Each should be a Ti*k*Z example demonstrating how the key is used.

Note that you can take advantage of the Lua syntax of enclosing very long multi-line strings in `[[` and `]]`. As a bonus, if the summary, documentation, or an example starts and ends with a quote, these two quotes will be stripped. This allows you to enclose the whole multi-line string (additionally) in quotes, leading to better syntax highlighting in editors.

External Documentation. It is sometimes more desirable to put the documentation of a key into an external file. First, this makes the code leaner and, thus, faster to read (both for humans and for computers). Second, for C code, it is quite inconvenient to have long strings inside a C file. In such cases, you can use the `documentation_in` field:

```
---
declare {
  key     = "electrical charge",
  type    = "number",
  initial = "1.0",
  documentation_in = "some_filename"
}
```

The `some_filename` must be the name of a Lua file that will be read "on demand", that is, whenever someone tries to access the documentation, summary, or examples field of the key, this file will be loaded using `require`. The file should then use `pgf.gd.doc` to install the missing information in the keys.

The Use Field. When you declare a key, you can provide a `use` field. If present, you must set it to an array of small tables which have two fields:

- `key` This is the name of another key or a function.
- `value` This is either a value (like a string or a number) or a function or `nil`.

Here is an example:

```
---
declare {
  key = "binary tree layout",
  use = {
    { key = "minimum number of children", value = 2 },
    { key = "significant sep",            value = 12 },
    { key = "tree layout" }
  },
  summary = "The |binary tree layout| places node...",
  documentation = ...,
  examples = ...,
}
```

The effect of a `use` field is the following: Whenever the key is encountered on the option stack, the key is first handled normally. Then, we iterate over all elements of the `use` array. For each element, we perform the action as if the `key` of the array had been set explicitly to the value given by the `value` field. If the `value` is a function, we pass a different value to the key, namely the result of applying the function to the value originally passed to the original key. Here is a typical example:

```
---
declare {
  key = "level sep",
  type = "length",
  use = {
    { key = "level pre sep",  value = function (v) return v/2 end },
    { key = "level post sep", value = function (v) return v/2 end }
  },
  summary = "..."
}
```

Just like the value, the key itself can also be a function. In this case, the to-be-used key is also computed by applying the function to the value passed to the original key.

As mentioned at the beginning, `declare` is a work-horse that will call different internal functions depending on whether you declare a parameter key or a new algorithm or a collection kind. Which kind of declaration is being done is detected by the presence of certain fields in the table passed to `t`. The different kind of possible declarations are documented in the `declare_...` functions. Note that these functions are internal and cannot be called from outside; you must use the `declare` function.

Parameters: 1. `t` A table contain the field `key` and other fields as described.

`function declare_parameter(t)`

This function is called by `declare` for "normal parameter keys", which are all keys for which no special field like `algorithm` or `layer` is declared. You write

```
---
declare {
  key     = "electrical charge",
  type    = "number",
  initial = "1.0",

  summary = "The ``electrical charge'' is a property...",
  documentation = [[...]],
  examples = [[...]]
}
```

When an author writes `my node[electrical charge=5-3]` in the description of her graph, the object `vertex` corresponding to the node `my node` will have a field `options` attached to it with

```
vertex.options["electrical charge"] == 2
```

The `type` field does not refer to Lua types. Rather, these types are sensible types for graph drawing and they are mapped by the higher layers to Lua types. In detail, the following types are available:

- `number` A dimensionless number. Will be mapped to a normal Lua `number`. So, when the author writes `foo=5*2`, the `foo` key of the `options` field of the corresponding object will be set to `10.0`.
- `length` A "dimension" in the sense of TEX (a number with a dimension like `cm` attached to it). It is the job of the display layer to map this to a number in "TEX points", that is, to a multiple of 1/72.27th of an inch.

- `time` A "time" in the sense of `\pgfparsetime`. Examples are `6s` or `0.1min` or `6000ms`, all of which will map to `6`.
- `string` Some text. Will be mapped to a Lua `string`.
- `canvas coordinate` A position on the canvas. Will be mapped to a `model.Coordinate`.
- `boolean` A Boolean value.
- `raw` Some to-be-executed Lua text.
- `direction` Normally, an angle; however, the special values of `down`, `up`, `left`, `right` as well as the directions `north`, `north west`, and so on are also legal on the display layer. All of them will be mapped to a number. Furthermore, a vertical bar (`|`) will be mapped to `-90` and a minus sign (`-`) will be mapped to `0`.
- `hidden` A key of this type "cannot be set", that is, users cannot set this key at all. However algorithms can still read this key and, through the use of `alias`, can use the key as a handle to another key.
- `user value` The key stores a Lua user value (userdata). Such keys can only be set from C since user values cannot be created in Lua (let alone in Ti*k*Z).

If the `type` field is missing, it is automatically set to `"string"`.

A parameter can have an `initial` value. This value will be used whenever the parameter has not been set explicitly for an object.

A parameter can have a `default` value. This value will be used as the parameter value whenever the parameter is explicitly set, but no value is provided. For a key of type `"boolean"`, if no `default` is provided, `"true"` will be used automatically.

A parameter can have an `alias` field. This field must be set to the name of another key or to a function. Whenever you access the current key and this key is not set, the `alias` key is tried instead. If it is set, its value will be returned (if the `alias` key has itself an alias set, this is tried recursively). If the alias is not set either and neither does it have an initial value, the `initial` value is used. Note that in case the alias has its `initial` field set, the `initial` value of the current key will never be used.

The main purpose of the current key is to allow algorithms to introduce their own terminology for keys while still having access to the standard keys. For instance, the `OptimalHierarchyLayout` class uses the name `layerDistance` for what would be called `level distance` in the rest of the graph drawing system. In this case, we can declare the `layerDistance` key as follows:

```
declare {
  key     = "layerDistance",
  type    = "length",
  alias   = "level distance"
}
```

Inside the algorithm, we can write `...options.layerDistance` and will get the current value of the `level distance` unless the `layerDistance` has been set explicitly. Indeed, we might set the `type` to `hidden` to ensure that *only* the `level distance` can and must set to set the layerDistance.

Note that there is a difference between `alias` and the `use` field: Suppose we write

```
declare {
  key     = "layerDistance",
  type    = "length",
  use     = {
    { key = "level distance", value = lib.id }
  }
}
```

Here, when you say `layerDistance=1cm`, the `level distance` itself will be modified. When the `level distance` is set, however, the `layerDistance` will not be modified.

If the alias is a function, it will be called with the option table as its parameter. You can thus say things like

```
declare {
  key      = "layerDistance",
  type     = "length",
  alias    = function (option)
               return option["layer pre dist"] + option["layer post dist"]
             end
}
```

As a special courtesy to C code, you can also set the key `alias_function_string`, which allows you to put the function into a string that is read using `loadstring`.

(You cannot call this function directly, it is included for documentation purposes only.)

Parameters: 1. `t` The table originally passed to `declare`.

`function` declare_algorithm(t)

This function is called by `declare` for "algorithm keys". These keys are normally used without a value as in just `\graph[tree layout]`, but you can optionally pass a value to them. In this case, this value must be the name of a *phase* and the algorithm of this phase will be set (and not the default phase of the key), see the description of phases below for details.

Algorithm keys are detected by the presence of the field `algorithm` in the table `t` passed to `declare`. Here is an example of how it is used:

```
local ReingoldTilford1981 = {}

---
declare {
  key       = "tree layout",
  algorithm = ReingoldTilford1981,

  preconditions = {
    connected = true,
    tree      = true
  },

  postconditions = {
    upward_oriented = true
  },

  summary = "The Reingold--Tilford method is...",
  documentation = ...,
  examples = ...,
}

function ReingoldTilford1981:run()
  ...
end
```

The `algorithm` field expects either a table or a string as value. If you provide a string, then `require` will be applied to this string to obtain the table; however, this will happen only when the key is actually used for the first time. This means that you can declare (numerous) algorithms in a library without these algorithms actually being loaded until they are needed.

Independently of how the table is obtained, it will be "upgraded" to a class by setting its `__index` field and installing a static `new` function (which takes a table of initial values as argument). Both these settings will only be done if they have not yet been performed.

Next, you can specify the fields `preconditions` and `postconditions`. The preconditions are a table that tell the graph drawing engine what kind of graphs your algorithm expects. If the input graph is not of this kind, it will be automatically transformed to meet this condition. Similarly, the postconditions tell the engine about properties of your graph after the algorithm has run. Again, additional transformations may be performed.

You can also specify the field `phase`. This tells the graph drawing engine which "phase" of the graph drawing process your option applies to. Each time you select an algorithm later on through use of the algorithm's key, the algorithm for this phase will be set; algorithms of other phases will not be changed. For instance, when an algorithm is part of the spanning tree computation, its phase will be `"spanning tree computation"` and using its key does not change the main algorithm, but only the algorithm used during the computation of a spanning tree for the current graph (in case this is needed

by the main algorithm). In case the `phase` field is missing, the phase `main` is used. Thus, when no phase field is given, the key will change the main algorithm used to draw the graph.

Later on, the algorithm set for the current phase can be accessed through the special `algorithm_phases` field of `options` tables. The `algorithm_phases` table will contain two fields for each phase for which some algorithm has been set: One field is the name of the phase and its value will be the most recently set algorithm (class) set for this phase. The other field is the name of the phase followed by `" stack"`. It will contain an array of all algorithm classes that have been set for this key with the most recently at the end.

The following example shows the declaration of an algorithm that is the default for the phase `"spanning tree computation"`:

```
---
declare {
  key = "breadth first spanning tree",
  algorithm = {
    run =
      function (self)
        return SpanningTreeComputation.computeSpanningTree(self.ugraph, false, self.events)
      end
  },
  phase = "spanning tree computation",
  phase_default = true,
  summary = ...
}
```

The algorithm is called as follows during a run of the main algorithms:

```
local graph = ... -- the graph object
local spanning_algorithm_class = graph.options.algorithm_phases["spanning tree computation"]
local spanning_algorithm =
  spanning_algorithm_class.new{
    ugraph = ugraph,
    events = scope.events
  }
local spanning_tree = spanning_algorithm:run()
```

If you set the `phase_default` field of `t` to `true`, the algorithm will be installed as the default algorithm for the phase. This can be done only once per phase. Furthermore, for such a default algorithm the `algorithm` key must be table, it may not be a string (in other words, all default algorithms are loaded immediately). Accessing the `algorithm_phases` table for a phase for which no algorithm has been set will result in the default algorithm and the phase stack will also contain this algorithm; otherwise the phase stack will be empty.

(You cannot call this function directly, it is included for documentation purposes only.)

Parameters: 1. `t` The table originally passed to `declare`.

`function` declare_collection_kind(t)

This function is called by `declare` for "collection kinds". They are detected by the presence of the field `layer` in the table `t` passed to `declare`. See the class `Collection` for details on what a collection and a collection kind is.

The `key` field of the table `t` passed to this function is both the name of the to-be-declared collection kind as well as the key that is used on the display layer to indicate that a node or edge belongs to a collection.

The Display Layer. Let us first have a look at what happens on the display layer: A key `t.key` is setup on the display layer that, when used inside a graph drawing scope, starts a new collection of the specified kind. "Starts" means that all nodes and edges mentioned in the rest of the current option scope will belong to a new collection of kind `t.key`.

```
declare { key = "hyper", layer = 1 }
```

you can say on the Ti*k*Z layer

```
\graph {
  a, b, c, d;
  { [hyper] a, b, c }
  { [hyper] b, c, d }
};
```

In this case, the nodes `a`, `b`, `c` will belong to a collection of kind `hyper`. The nodes `b`, `c`, and `d` will (also) belong to another collection of the same kind `hyper`. You can nest collections; in this case, nodes will belong to several collections.

The effect of declaring a collection kind on the algorithm layer it, first of all, that `scope.collections` will have a field named by the collection kind. This field will store an array that contains all collections that were declared as part of the graph. For instance, `collections.hyper` will contain all hyperedges, each of which is a table with the following fields: The `vertices` and `edges` fields each contain arrays of all objects being part of the collection. The `sub` field is an array of "subcollections", that is, all collections that were started inside another collection. (For the collection kinds `hyper` and `same layer` this makes no sense, but subgraphs could, for instance, be nested.)

Rendering of Collections. For some kinds of collections, it makes sense to *render* them, but only after the graph drawing algorithm has run. For this purpose, the binding layer will use a callback for each collection kind and each collection, see the `Binding` class for details. Suppose, for instance, you would like hyperedges to be rendered. In this case, a graph drawing algorithm should iterate over all collections of type `hyper` and compute some hints on how to render the hyperedge and store this information in the `generated_options` table of the hyperedge. Then, the binding layer will ask the display layer to run some some code that is able to read key–value pairs passed to it (which are the key–value pairs of the `generated_options` table) and use this information to nicely draw the hyperedge.

The number `t.layer` determines in which order the different collection kinds are rendered.

The last parameter, the layer number, is used to specify the order in which the different collection kinds are rendered. The higher the number, the later the collection will be rendered. Thus, if there is a collection kind with layer number 10 and another with layer number 20, all collections of the first kind will be rendered first, followed by all collections of the second kind.

Collections whose layer kinds are non-negative get rendered *after* the nodes and edges have already been rendered. In contrast, collections with a negative layer number get shown "below" the nodes and edges.

(You cannot call this function directly, it is included for documentation purposes only.)

Parameters: 1. `t` The table originally passed to `declare`.

`function InterfaceToAlgorithms.findVertexByName(name)`

Finds a node by its name. This method should be used by algorithms for which a node name is specified in some option and, thus, needs to be converted to a vertex object during a run of the algorithm.

Parameters: 1. `name` A node name

Returns: 1. The vertex of the given name in the syntactic digraph or `nil`.

`function InterfaceToAlgorithms.createVertex(algorithm, init)`

Generate a new vertex in the syntactic digraph. Calling this method allows algorithms to create vertices that are not present in the original input graph. Using the graph drawing coroutine, this function will pass back control to the display layer in order to render the vertex and, thereby, create precise size information about it.

Note that creating new vertices in the syntactic digraph while the algorithm is already running is a bit at odds with the notion of treating graph drawing as a series of graph transformations: For instance, when a new vertex is created, the graph will (at least temporarily) no longer be connected; even though an algorithm may have requested that it should only be fed connected graphs. Likewise, more complicated requirements like insisting on the graph being a tree also cannot be met.

For these reasons, the following happens, when a new vertex is created using the function:

1. The vertex is added to the syntactic digraph.
2. It is added to all layouts on the current layout stack. When a graph drawing algorithm is run, it is not necessarily run on the original syntactic digraph. Rather, a sequence / stack of nested layouts may currently be processed and the vertex is added to all of them.

3. The vertex is added to both the `digraph` and the `ugraph` of the current algorithm.

Parameters: *1.* `algorithm` An algorithm for whose syntactic digraph the node should be added *2.* `init` A table of initial values for the node that is passed to `Binding:createVertex`, see that function for details.

Returns: *1.* The newly created node

`function InterfaceToAlgorithms.createEdge(algorithm, tail, head, init)`

Generate a new edge in the syntactic digraph. This method is quite similar to `createVertex` and has the same effects with respect to the edge: The edge is added to the syntactic digraph and also to all layouts on the layout stack. Furthermore, appropriate edges are added to the `digraph` and the `ugraph` of the algorithm currently running.

Parameters: *1.* `algorithm` An algorithm for whose syntactic digraph the node should be added *2.* `tail` A syntactic tail vertex *3.* `head` A syntactic head vertex *4.* `init` A table of initial values for the edge.

The following fields are useful for `init`:

- `init.direction` If present, a direction for the edge. Defaults to "–".
- `init.options` If present, some options for the edge.
- `init.generated_options` A table that is passed back to the display layer as a list of key-value pairs in the syntax of `declare_parameter`.

36.8 Examples of Implementations of Graph Drawing Algorithms

Graph Drawing Library `examples`

```
\usegdlibrary{examples} % LaTeX and plain TeX
\usegdlibrary[examples] % ConTeXt
```

This package presents some examples of how different aspects of the graph drawing engine can be used. In particular, the algorithms of this package are not really meant to be used to layout graphs (although they can be used, in principle); rather you are invited to have a look at their implementation and to adapt them to your needs.

36.8.1 The "Hello World" of Graph Drawing

`/graph drawing/simple demo layout=`⟨*string*⟩

This algorithm is the "Hello World" of graph drawing. The algorithm arranges nodes in a circle (without paying heed to the sizes of the nodes or to the edges). In order to "really" layout nodes in a circle, use `simple necklace layout`; the present layout is only intended to demonstrate how much (or little) is needed to implement a graph drawing algorithm.

```
-- File pgf.gd.examples.SimpleDemo
local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare

declare {
  key = "simple demo layout",
  algorithm = {
    run =
      function (self)
        local g = self.digraph
        local alpha = (2 * math.pi) / #g.vertices

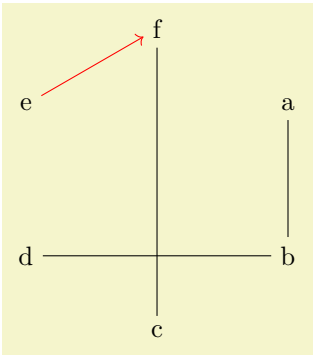
        for i,vertex in ipairs(g.vertices) do
          local radius = vertex.options['radius'] or g.options['radius']
          vertex.pos.x = radius * math.cos(i * alpha)
          vertex.pos.y = radius * math.sin(i * alpha)
        end
      end
  },
  summary = "This algorithm is the 'Hello World' of graph drawing.",
  documentation = [["
    This algorithm arranges nodes in a circle ...
  "]]
}
```

On the display layer (Ti*k*Z, that is) the algorithm can now immediately be employed; you just need to say `\usegdlibrary{examples.SimpleDemo}` at the beginning somewhere.

36.8.2 How To Generate Edges Inside an Algorithm

`/graph drawing/simple edge demo layout=`⟨*string*⟩

This algorithm shows how edges can be created by an algorithm. For its job, the algorithm uses the function `createEdge`, which can be called during the run of the algorithm to create edges in the syntactic graph. The algorithm first does exactly the same as the simple demo layout, then it creates an edge for every node where the `new edge to` option is set. You will see in the code how this option is declared and how we use it to look up a vertex in the graph by its name.




```
\usetikzlibrary {graphs,graphdrawing} \usegdlibrary {examples}
\tikz [simple edge demo layout]
\graph [radius=2cm] {
  a, b, c, d, e, f;

  e -> [red] f; % normal edge

  % Edges generated by the algorithm:
  a[new edge to=b];
  b[new edge to=d];
  c[new edge to=f];
};
```

And the algorithm:

```
-- File pgf.gd.examples.SimpleEdgeDemo

-- Imports
local InterfaceToAlgorithms = require "pgf.gd.interface.InterfaceToAlgorithms"
local declare               = InterfaceToAlgorithms.declare

-- The class object
local SimpleEdgeDemo = {}

declare {
  key = "simple edge demo layout",
  algorithm = SimpleEdgeDemo,
  summary = "This algorithm shows...",
}
```

Next comes the declaration of the new option `new edge to`:

```
declare {
  key = "new edge to",
  type = "string",
  summary = "This option takes the name of a vertex..."
}
```

Finally, the algorithm's code:

```
function SimpleEdgeDemo:run()
  -- As in a SimpleDemo:
  ...
  -- Now add some edges:
  for _,tail in ipairs(g.vertices) do
    local name = tail.options['new edge to']
    if name then
      local node = InterfaceToAlgorithms.findVertexByName(name)
      if node and self.digraph:contains(node) then
        InterfaceToAlgorithms.createEdge (self, tail, node)
      end
    end
  end
end
```

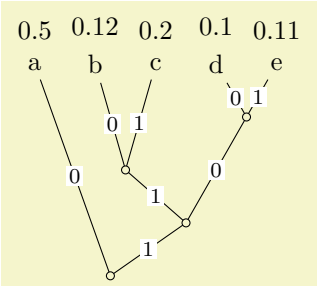
/graph drawing/new edge to=⟨*string*⟩

This option takes the name of a vertex. An edge leading to this vertex is added to the syntactic digraph.

36.8.3 How To Generate Nodes Inside an Algorithm

/graph drawing/simple Huffman layout=⟨*string*⟩

This algorithm demonstrates how an algorithm can generate new nodes. The input graph should just consist of some nodes (without edges) and each node should have a `probability` key set. The nodes will then be arranged in a line (as siblings) and a Huffman tree will be constructed "above" these nodes. For the construction of the Huffman tree, new nodes are created and connected.




```
\usetikzlibrary {graphs,graphdrawing,quotes} \usegdlibrary {examples}
\tikz \graph [simple Huffman layout,
              level distance=7mm, sibling distance=8mm, grow'=up]
{
  a ["0.5",  probability=0.5],
  b ["0.12", probability=0.12],
  c ["0.2",  probability=0.2],
  d ["0.1",  probability=0.1],
  e ["0.11", probability=0.11]
};
```

The file starts with some setups and declarations:

```
-- File pgf.gd.examples.SimpleHuffman

local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare

-- The class
local SimpleHuffman = {}

declare {
  key            = "simple Huffman layout",
  algorithm      = SimpleHuffman,
  postconditions = { upward_oriented = true  }
  summary = "..."
}

declare {
  key = "probability",
  type = "number",
  initial = "1",
  summary = "..."
}

-- Import
local layered = require "pgf.gd.layered"
local InterfaceToAlgorithms = require "pgf.gd.interface.InterfaceToAlgorithms"
local Storage = require "pgf.gd.lib.Storage"

local probability = Storage.new()
local layer       = Storage.new()

function SimpleHuffman:run()
  -- Construct a Huffman tree on top of the vertices...
```

Next comes a setup, where we create the working list of vertices that changes as the Huffman coding method proceeds:

```
  -- Shorthand
  local function prop (v)
    return probability[v] or v.options['probability']
  end

  -- Copy the vertex table, since we are going to modify it:
  local vertices = {}
  for i,v in ipairs(self.ugraph.vertices) do
    vertices[i] = v
  end
```

The initial vertices are arranged in a line on the last layer. The function `ideal_sibling_distance` takes care of the rather complicated handling of the (possibly rotated) bounding boxes and separations. The `props` and `layer` are tables used by algorithms to "store stuff" at a vertex or at an arc. The table will be accessed by `arrange_layers_by_baselines` to determine the ideal vertical placements.

```
  -- Now, arrange the nodes in a line:
  vertices [1].pos.x = 0
  layer[ vertices [1] ] = #vertices
  for i=2,#vertices do
    local d = layered.ideal_sibling_distance(self.adjusted_bb, self.ugraph, vertices[i-1], vertices[i])
    vertices [i].pos.x = vertices[i-1].pos.x + d
    layer[ vertices [i] ] = #vertices
  end
```

Now comes the actual Huffman algorithm: Always find the vertices with a minimal probability…

```
  -- Now, do the Huffman thing...
  while #vertices > 1 do
    -- Find two minimum probabilities
    local min1, min2

    for i=1,#vertices do
      if not min1 or prop(vertices[i]) < prop(vertices[min1]) then
        min2 = min1
        min1 = i
      elseif not min2 or prop(vertices[i]) < prop(vertices[min2]) then
        min2 = i
      end
    end
```

…and connect them with a new node. This new node gets the option `HuffmanNode`. It is now the job of the higher layers to map this option to something "nice".

```
    -- Create new node:
    local p = prop(vertices[min1]) + prop(vertices[min2])
    local v = InterfaceToAlgorithms.createVertex(self, { generated_options = {{key="HuffmanNode"}}})
    probability[v] = p
    layer[v] = #vertices-1
    v.pos.x = (vertices[min1].pos.x + vertices[min2].pos.x)/2
    vertices[#vertices + 1] = v

    InterfaceToAlgorithms.createEdge (self, v, vertices[min1],
        {generated_options = {{key="HuffmanLabel", value = "0"}}})
    InterfaceToAlgorithms.createEdge (self, v, vertices[min2],
        {generated_options = {{key="HuffmanLabel", value = "1"}}})

    table.remove(vertices, math.max(min1, min2))
    table.remove(vertices, math.min(min1, min2))
  end
```

Ok, we are mainly done now. Finish by computing vertical placements and do formal cleanup.

```
  layered.arrange_layers_by_baselines(layers, self.adjusted_bb, self.ugraph)
end
```

In order to use the class, we have to make sure that, on the display layer, the options `HuffmanLabel` and `HuffmanNode` are defined. This is done by adding, for instance, the following to Ti*k*Z:

```
\pgfkeys{
  /graph drawing/HuffmanLabel/.style={
    /tikz/edge node={node[fill=white,font=\footnotesize,inner sep=1pt]{#1}}
  },
  /graph drawing/HuffmanNode/.style={
    /tikz/.cd,circle,inner sep=0pt,outer sep=0pt,draw,minimum size=3pt
  }
}
```

`/graph drawing/probability=`⟨*number*⟩ (initially `1`)

The probability parameter. It is used by the Huffman algorithm to group nodes.

36.9 Support Libraries

The present section lists a number of general-purpose libraries that are used by different algorithms.

36.9.1 Basic Functions

Lua table `pgf` (declared in `pgf`)

The `pgf` namespace lies in the global namespace. It is the only global table defined by PGF. The whole graph drawing system, in turn, lies in the table `pgf.gd`.

`function pgf.debug(...)`

Writes debug info on the TEX output, separating the parameters by spaces. The debug information will include a complete traceback of the stack, allowing you to see "where you are" inside the Lua program.

Note that this function resides directly in the `pgf` table. The reason for this is that you can "always use it" since `pgf` is always available in the global name space.

Parameters: *1.* `...` List of parameters to write to the TEX output.

Lua table `lib` (declared in `pgf.gd.lib`)

Basic library functions

Alphabetical method summary:

`function lib.class (t)`

`function lib.copy (source, target)`

`function lib.find_min (array, f)`

`function lib.find (array, test)`

`function lib.icopy (source, target)`

`function lib.id (...)`

`function lib.imap (source, fun, new)`

`function lib.lookup_option (name, ...)`

`function lib.map (source, fun)`

`function lib.ondemand (filename, table, name)`

`function lib.random_permutation (n)`

`function lib.randomseed (seed)`

`function lib.random (l,u)`

`function lib.find(array, test)`

Finds the first value in the `array` for which `test` is true.

Parameters: *1.* `array` An array to search in. *2.* `test` A function that is applied to each element of the array together with the index of the element and the whole table.

Returns: *1.* The value of the first value where the test is true.

Returns: *1.* The index of the first value where the test is true.

Returns: *1.* The function value of the first value where the test is true (only returned if test is a function).

`function lib.find_min(array, f)`

Finds the first value in the `array` for which a function returns a minimal value

Parameters: *1.* `array` An array to search in. *2.* `f` A function that is applied to each element of the array together with the index of the element and the whole table. It should return an integer and, possibly, a value.

Among all elements for which a non-nil integer is returned, let `i` by the index of the element where this integer is minimal.

Returns: *1.* `array[i]`

Returns: *1.* `i`

Returns: *1.* The return value(s) of the function at `array[i]`.

`function lib.copy(source, target)`

Copies a table while preserving its metatable.

Parameters: *1.* `source` The table to copy. *2.* `target` The table to which values are to be copied or `nil` if a new table is to be allocated.

Returns: *1.* The `target` table or a newly allocated table containing all keys and values of the `source` table.

`function lib.icopy(source, target)`

Copies an array while preserving its metatable.

Parameters: *1.* `source` The array to copy. *2.* `target` The array to which values are to be copied or `nil` if a new table is to be allocated. The elements of the `source` array will be added at the end.

Returns: *1.* The `target` table or a newly allocated table containing all keys and values of the `source` table.

`function lib.map(source, fun)`

Apply a function to all pairs of a table, resulting in a new table.

Parameters: *1.* `source` The table. *2.* `fun` A function taking two arguments (`val` and `key`, in that order). Should return two values (a `new_val` and a `new_key`). This pair will be inserted into the new table. If, however, `new_key` is `nil`, the `new_value` will be inserted at the position `key`. This means, in particular, that if the `fun` takes only a single argument and returns only a single argument, you have a "classical" value mapper. Also note that if `new_value` is `nil`, the value is removed from the table.

Returns: *1.* The new table.

`function lib.imap(source, fun, new)`

Apply a function to all elements of an array, resulting in a new array.

Parameters: *1.* `source` The array. *2.* `fun` A function taking two arguments (`val` and `i`, the current index). This function is applied to all elements of the array. The result of this function is placed at the end of a new array, expect when the function returns `nil`, in which case the element is skipped. If this function is not provided (is `nil`), the identity function is used. *3.* `new` The target array (if `nil`, a new array is create).

```
local a = lib.imap(array, function(v) if some_test(v) then return v end end)
```

The above code is a filter that will remove all elements from the array that do not pass `some_test`.

```
lib.imap(a, lib.id, b)
```

The above code has the same effect as `lib.icopy(a,b)`.

Returns: *1.* The new array

`function lib.random_permutation(n)`

Generate a random permutation of the numbers 1 to n in time $O(n)$. Knuth's shuffle is used for this.

Parameters: *1.* `n` The desired size of the table

Returns: *1.* A random permutation

`function lib.id(...)`

The identity function, so you can write `lib.id` instead of `function (x) return x end`.

`function lib.lookup_option(name, ...)`

Tries to find an option in different objects that have an options field.

This function iterates over all objects given as parameters. In each, it tries to find out whether the options field of the object contains the option `name` and, if so, returns the value. The important point is that checking whether the option table of an object contains the name field is done using `rawget` for all but the last parameter. This means that when you write

```
lib.lookup_option("foo", vertex, graph)
```

and if `/graph drawing/foo` has an initial value set, if the parameter is not explicitly set in a vertex, you will get the value set for the graph or, if it is not set there either, the initial value. In contrast, if you write

```
 vertex.options["foo"] or graph.options["foo"]
```

what happens is that the first access to `.options` will *always* return something when an initial parameter has been set for the option `foo`.

Parameters: *1.* `name` The name of the options *2.* `...` Any number of objects. Each must have an options field.

Returns: *1.* The found option

`function lib.class(t)`

Turns a table `t` into a class in the sense of object oriented programming. In detail, this means that `t` is augmented by a `new` function, which takes an optional table of `initial` values and which outputs a new table whose metatable is the class. The `new` function will call the function `constructor` if it exists. Furthermore, the class object's `__index` is set to itself and its meta table is set to the `base_class` field of the table. If `t` is `nil`, a new table is created.

Here is a typical usage of this function:

```
local Point = lib.class {}

function Point:length()
  return math.sqrt(self.x*self.x + self.y*self.y)
end

local p = Point.new { x = 5, y = 6 }

print(p:length())
```

We can subclass this as follows:

```
local Point3D = lib.class { base_class = Point }

function Point3D:length()
  local l = Point.length(self) -- Call base class's function
  return math.sqrt(l*l + self.z*self.zdy)
end

local p = Point3D.new { x = 5, y = 6, z = 6 }

print(p:length())
```

Parameters: *1.* `t` A table that gets augmented to a class. If `nil`, a new table is created.

Returns: *1.* The augmented table.

`function lib.ondemand(filename, table, name)`

Returns a method that is loaded only on demand for a class.

The idea behind this function is that you may have a class (or just a table) for which some methods are needed only seldomly. In this case, you can put these methods in a separate file and then use `ondemand` to indicate that the methods are found in a another file.

```
-- File Foo.lua
local Foo = {}
function Foo.bar ()  ... end
function Foo.bar2 () ... end
Foo.bar3 = lib.ondemand("Foo_extra", Foo, "bar3")
Foo.bar4 = lib.ondemand("Foo_extra", Foo, "bar4")

return Foo

-- Foo_extra.lua
local Foo = require "Foo"
function Foo.bar3 () ... end
function Foo.bar4 () ... end
```

Parameters: *1.* `filename` The name of the file when extra methods are located. *2.* `table` The table for which the missing functions should be loaded when they are accessed. *3.* `method` The name of the method.

Returns: *1.* A function that, when called, loads the filename using `require` and, then, forwards the call to the method.

`function lib.random(l,u)`

This implements the a random number generator similar to the one provided by Lua, but based on the tex.uniformdeviate primitive to avoid differences in random numbers due to platform specifics.

Parameters: *1.* `l` Lower bound *2.* `u` Upper bound

Returns: *1.* A random number

`function lib.randomseed(seed)`

Provide the seed for the random number generator

Parameters: *1.* `seed` random seed

36.9.2 Lookup Tables

Lua table `LookupTable` (declared in `pgf.gd.lib.LookupTable`)

This table provides two utility functions for managing "lookup tables". Such a table is a mixture of an array and a hashtable: It stores (only) tables. Each table is stored once in a normal array position. Additionally, the lookup table is also indexed at the position of the table (used as a key) and this position is set to `true`. This means that you can test whether a table `t` is in the lookup table `l` simply by testing whether `l[t]` is true.

Alphabetical method summary:

`function LookupTable.addOne (l, e)`

`function LookupTable.add (l, array)`

`function LookupTable.remove (l, array)`

`function LookupTable.add(l, array)`

Add all elements in the `array` to a lookup table. If an element of the array is already present in the table, it will not be added again.

This operation takes time $O(|\texttt{array}|)$.

Parameters: *1.* `l` Lookup table *2.* `array` An array of to-be-added tables.

`function LookupTable.addOne(l, e)`

Add one element to a lookup table. If it is already present in the table, it will not be added again.

This operation takes time $O(1)$.

Parameters: *1.* `l` Lookup table *2.* `e` The to-be-added element.

`function LookupTable.remove(l, array)`

Remove tables from a lookup table.

Note that this operation is pretty expensive insofar as it will always cost a traversal of the whole lookup table. However, this is also the maximum cost, even when a lot of entries need to be deleted. Thus, it is much better to "pool" multiple remove operations in a single one.

This operation takes time $O(\max\{|\texttt{array}|, |\texttt{l}|\})$.

Parameters: 1. `l` Lookup table *2.* `t` An array of to-be-removed tables.

36.9.3 Computing Distances in Graphs

Still needs to be ported to digraph classes!

36.9.4 Priority Queues

Lua table `PriorityQueue` (declared in `pgf.gd.lib.PriorityQueue`)

A PriorityQueue supports operations for quickly finding the minimum from a set of elements

Its implementation is based on (simplified) Fibonacci heaps.

Alphabetical method summary:

```
function PriorityQueue.new ()
function PriorityQueue:dequeue ()
function PriorityQueue:enqueue (value, priority)
function PriorityQueue:isEmpty ()
function PriorityQueue:updatePriority (value, priority)
```

`function PriorityQueue.new()`

Creates a new priority queue

Returns: 1. The newly created queue

`function PriorityQueue:enqueue(value, priority)`

Add an element with a certain priority to the queue

Parameters: 1. `value` An object *2.* `priority` Its priority

`function PriorityQueue:dequeue()`

Removes the element with the minimum priority from the queue

Returns: 1. The element with the minimum priority

`function PriorityQueue:updatePriority(value, priority)`

Lower the priority of an element of a queue

Parameters: 1. `value` An object *2.* `priority` A new priority, which must be lower than the old priority

`function PriorityQueue:isEmpty()`

Tests, whether the queue is empty

Returns: 1. True, if the queue is empty

37 Writing Graph Drawing Algorithms in C

by Till Tantau

In the present section we have a look at how graph drawing algorithms written in the C programming language (or in C++) can be used in the graph drawing framework.

> *Warning:* Graph drawing algorithms written in C can be incredibly fast if you use the facilities of C correctly. *However,* C code is much less portable than Lua code in the sense that it has to be compiled for the specific platform used by the user and that it has to be linked dynamically during a run of the TEX program. All of this in possible (and works, as demonstrated by the linking of the OGDF framework), but it is *much* harder to get right than writing Lua code.
>
> Bottom line, *you really should be using this method only if it is really necessary (namely, when Lua code is simply not fast enough).*

In the following, I first explain how the link between TEX and C code works, in general. Then, in the subsequent sections, we go over the different kinds of programming languages and frameworks for which there is direct support for such a link.

37.1 How C and TEX Communicate

In order to use C code for graph drawing algorithms during a run of the TEX program, there is no need to build a new version of TEX. Rather, it is possible that C code is linked into the TEX executable at runtime. This is made possible by the fact that Lua (which part of LuaTEX...) is able to link C libraries at runtime – provided a strict regime of rules is adhered to:

1. When you say `require` in Lua, it will normally look for a `.lua` file; but it will also try to find a `.so` file (a shared C library) as a fallback.
2. If it finds such a shared library, Lua(TEX) will try to link this library dynamically at runtime.
3. Inside the library, there must be a function (called an entry point) with a special name (it must start with `luaopen_` and it must otherwise be the path and name of the library with slashes replaced by underscores).
4. This function gets called by Lua, once. Its job is to setup the library so that it can be used by Lua. Mainly, this means that certain C functions get registered in such a way that Lua can call them.
5. At this point, control returns to Lua and, now, certain functions have become available on the Lua layer that, when called, actually invoke the C code of our linked library.

For each of the above points, there are some bells and whistles:

1. LuaTEX looks at slightly inconvenient places for shared libraries: By default, (currently, 2013) it looks in a `lib` subdirectory of the directory containing the LuaTEX executable. The logic behind is that the shared libraries depend on the specific architecture of the executable. Thus, unlike normal Lua files, the library needs to be installed "far away" from the actual package of which it is part.
2. Certain versions of LuaTEX have a broken handling of filenames of libraries written in C. The TL2013 version of LuaTEX, for instance, crashes when the filename of a shared library does not contain the complete path (while this works for normal file). Hopefully, this, too, will be fixed in future versions.
3. On certain platforms, the support for dynamic linking against LuaTEX is broken since the symbol table of the Lua library has been stripped away. Hopefully, this will be fixed soon; in the meantime, a highly fragile workaround is to link in another copy of the Lua library.
4. The entry point that is called by Lua requires a certain signature (it takes a Lua state as its only parameter) and must return the number of objects it returns on the Lua stack.
5. The registration process of C functions is somewhat tricky and changes from Lua version to Lua version.
6. C functions that get called by Lua must follow all sorts of tricky rules in order to communicate with Lua correctly.

Despite the above obstacles, one can use graph drawing algorithms written in C inside Lua, in principle, as follows: One loads an appropriately prepared and located C library using `require` and this library uses commands like `declare` to register its own functions into the graph drawing system so that when the `run` method is called, a C functions gets called instead.

Unfortunately, the above approach is extremely tedious and error-prone and it is "no fun" to access Lua data structures (such as the syntactic digraph) from C. For this reason, I have written some libraries that encapsulate (as much as possible) of this communication between C and Lua. Indeed, when you use these libraries, you can focus entirely on the graph drawing issues and you will not even notice that your code "is talking to Lua". (Except for the name of the entry point, which is fixed to start with `luaopen_` and it is impossible to change this without disrupting a lot inside Lua's module system).

There are libraries available for simplifying the communication between the graph drawing system and graph drawing algorithms written in

- C, see Section 37.2,
- C++, see Section 37.3,
- Open Graph Drawing Framework, see Section 37.4.

37.2 Writing Graph Drawing Algorithms in C

37.2.1 The Hello World of Graph Drawing in C

As our first example, as always, the "hello world" of graph drawing simply places nodes on a circle. For this, we implement a function `fast_hello_world` in a file `SimpleDemoC.c`. It starts as follows:

```
#include <pgf/gd/interface/c/InterfaceFromC.h>
#include <math.h>

static void fast_hello_world (pgfgd_SyntacticDigraph* graph) {
  ...
}
```

As we can see, we first include a special header file of a rather small library that does all the hard work of translating between Lua and C for us (`InterfaceFromC`). These header files reside in the `c` subdirectory of the `pgf` package. Note that we do *not* have to include the headers of the Lua library; indeed, you do not need access to the source of Lua to use the interface headers. As a side effect, we will, however, have to write `struct lua_State` instead of the more common `lua_State` once in our code, namely in the declaration of the entry point; but that is the only downside.

The library `InterfaceFromC` declares the type `pgfgd_SyntacticDigraph`. In a moment, we will see that we can setup a key `fast simple demo layout` such that when this key is used on the display layer, the function `fast_hello_world` gets called. When it is called, the `graph` parameter will be a full representation of the to-be-laid-out graph. We can access the fields of the graph and even directly modify some of its fields (in particular, we can modify the `pos` fields of the vertices). Here is the complete code of the algorithm:

```
static void fast_hello_world (pgfgd_SyntacticDigraph* graph) {
  double angle  = 6.28318530718 / graph->vertices.length;
  double radius = pgfgd_tonumber(graph->options, "fast simple demo radius");

  int i;
  for (i = 0; i < graph->vertices.length; i++) {
    pgfgd_Vertex* v = graph->vertices.array[i];
    v->pos.x = cos(angle*i) * radius;
    v->pos.y = sin(angle*i) * radius;
  }
}
```

That is all that is needed; the C library will take care of both creating the `graph` object as all well as of deleting it and of copying back the computed values of the `pos` fields of the vertices.

Our next task is to setup the key `fast simple demo layout`. We can (and must) also do this from C, using the following code:

```
int luaopen_pgf_gd_examples_c_SimpleDemoC (struct lua_State *state) {

  pgfgd_Declaration* d = pgfgd_new_key ("fast simple demo layout");
  pgfgd_key_summary          (d, "The C version of the hello world of graph drawing");
  pgfgd_key_algorithm        (d, fast_hello_world);
  pgfgd_key_add_precondition (d, "connected");
  pgfgd_key_add_precondition (d, "tree");
  pgfgd_declare              (state, d)
  pgfgd_free_key             (d);
```

The function `luaopen_pgf_gd_examples_c_SimpleDemoC` is the function that will be called by Lua (we will come to that). More important for us, at the moment, is the declaration of the key: We use `pgfgd_new_key` to create a declaration record and then fill the different fields using appropriate function calls. In particular, the call `pgfgd_key_algorithm` allows us to link the key with a particular C function. The `pgfgd_declare` will then pass the whole declaration back to Lua, so the effect of the above is essentially the same as if you had written in Lua:

```
declare {
  key = "fast simple demo layout",
  summary = "The C version of the hello world of graph drawing",
  preconditions = {
    connected = true,
    tree = true,
  },
  algorithm = {
    run =  -- something magic we cannot express in Lua
  }
}
```

In our algorithm, in addition to the above key, we also use the `fast simple demo radius` key, which is a simple length key. This key, too, can be declared on the C layer:

```
  d = pgfgd_new_key ("fast simple demo radius");
  pgfgd_key_summary (d, "A radius value for the hello world of graph drawing");
  pgfgd_key_type    (d, "length");
  pgfgd_key_initial (d, "1cm");
  pgfgd_declare     (state, d);
  pgfgd_free_key    (d);

  return 0;
}
```

We simply add this code to the startup function above.

Now it is time to compile and link the code. For this, you must, well, compile it, link it against the library `InterfaceFromC`, and build a shared library out of it. Also, you must place it somewhere where LuaTEX will find it. You will find a Makefile that should be able to achieve all of this in the directory `pgf/c/graphdrawing/pgf/gd/examples/c`, where you will also find the code of the above example.

Now, all you need to do to use it is to write in Lua (after you have loaded the `pgf.gd` library, of course), would normally be the call

```
require 'pgf.gd.examples.c.SimpleDemoC'
```

or in Ti*k*Z

```
\usegdlibrary {examples.c.SimpleDemoC}
```

This should cause LuaTEX to find the shared library, load it, and then call the function in that library with the lengthy name (the name is always `luaopen_` followed by the path and filename with slashes replaced by underscores).

Remark: Unfortunately, the above does not work with the TEX Live 2013 versions of LuaTEX due to a bugs that causes the "replace dots by slashes" to fail. For this reason, we currently need to rename our shared library file to

```
pgf_gd_examples_c_SimpleDemoC.so
```

and then say

```
require 'pgf_gd_examples_c_SimpleDemoC'
```

or in Ti*k*Z

```
\usegdlibrary {pgf_gd_examples_c_SimpleDemoC}
```

In future versions of LuaTEX, things should be "back to normal" in this regard. Also, the bug only concerns shared libraries; you can still create a normal Lua file with a nice name and place at a nice location and the only contents of this file is then the above `require` command.

Anyway, once we have loaded the shared library we can say:

```
\tikz \graph [fast simple demo layout, fast simple demo radius=1.25cm]
{ a -> b -> c -> d -> e -> a };
```

37.2.2 Documenting Algorithms Written in C

In our above example, we included a summary with the keys in the C code. It would be even better if we added a longer documentation and some examples that show how the key works; but this is a bit impracticable in C since multi-line strings are hard to write down in C. The trick is to use the `documentation_in` field of a key: It allows us to specify the name of a Lua file that should be loaded (using `require`) to install the missing documentation fields. As explained in Section 36.2.7, this Lua file may make good use the `pgf.gd.doc` package. Note, also, that for keys documented in this way the documentation can easily be included in this manual through the use of the `\includedocumentationof` command.

In our example, we would first add the following line twice in the C code (once for each key), assuming that the documentation resides in the file `pgf/gd/doc/examples/SimpleDemoC.lua`:

```
  pgfgd_key_documentation_in (d, "pgf.gd.doc.examples.SimpleDemoC");
```

Note that since the documentation is a normal Lua file, it will be searched in the usual places Lua files are located (in the texmf trees) and not, like the C shared library, in the special `lib` subdirectory of the LuaTEX binary.

Here are typical contents of the documentation file:

```
-- File pgf/gd/doc/examples/SimpleDemoC.lua
local key           = require 'pgf.gd.doc'.key
local documentation = require 'pgf.gd.doc'.documentation
local summary       = require 'pgf.gd.doc'.summary
local example       = require 'pgf.gd.doc'.example

key           "fast simple demo layout"
documentation
[[
This layout is used...
]]
example
[[
\tikz \graph [fast simple example layout]
{ a -- b -- c -- d -- e; };
]]

key           "fast simple demo radius"
documentation
[[
The radius parameter is used to ...
]]
example
[[
\tikz \graph [fast simple demo layout, fast simple demo radius=1.25cm]
{ a -> b -> c -> d -> e -> a };
]]
```

37.2.3 The Interface From C

In the above example, we already saw some of the functions from the library `InterfaceFromC` that translated from Lua to C for us. For a complete list of all functions available, currently please see `graphdrawing/c/pgf/gd/interface/c/InterfaceFromC.h` directly.

Currently, the library provides C functions to directly access all aspects of the syntactic digraph and also of the graphs computed by the preprocessing of the layout pipeline. What is missing, however, is access to the tree of (sub)layouts and to collections. Hopefully, these will be added in the future.

37.3 Writing Graph Drawing Algorithms in C++

Built on top of the C interface presented in the previous section, there is also a C++ interface available. It encapsulates as much of the C functions as possible in C++ classes. Thus, this interface is mostly there for convenience, it does not offer fundamentally new functionality.

37.3.1 The Hello World of Graph Drawing in C++

Let us have a look at how our beloved hello world of graph drawing looks in C++. Although it is still possible to put graph drawing algorithms inside functions, it is more natural in C++ to turn them into methods of a class. Thus, we start the code of `SimpleDemoCPlusPlus.c++` as follows:

```
#include <pgf/gd/interface/c/InterfaceFromC++.h>
#include <pgf/gd/interface/c/InterfaceFromC.h>

#include <math.h>

struct FastLayout : scripting::declarations, scripting::runner {
  ...
}
```

As can be seen, we do not only include the interface from C++, but also that from C (since, currently, not all functionality of the C library is encapsulated in C++).

The interesting part is the `struct FastLayout`, which will contain our algorithm (you could just as well have used a `class` instead of a `struct`). It is derived from two classes: First, from a `declarations` class and, secondly, from a `runner` class. Both of them, just like everything else from the interface, reside in the namespace `scripting`. This name was chosen since the main purpose of the interface is to provide "scripting facilities" to C code through the use of Lua.

We are currently interested in the class `runner`. This class has a virtual function `run` that gets called when, on the Lua side, someone has selected the algorithm represented by the class. Thus, we place our algorithm in this method:

```
void run () {
  pgfgd_SyntacticDigraph* graph = parameters->syntactic_digraph;

  double angle  = 6.28318530718 / graph->vertices.length;
  double radius = parameters->option<double>("fast simple demo radius c++");

  for (int i = 0; i < graph->vertices.length; i++) {
    pgfgd_Vertex* v = graph->vertices.array[i];
    v->pos.x = cos(angle*i) * radius;
    v->pos.y = sin(angle*i) * radius;
  }
}
```

The `run` method has access to the member variable `parameters`, which contains all sorts of information concerning the to-be-drawn graph. In particular, the `syntactic_digraph` field gives us access to the syntactic digraph structure that was already available in the interface from plain C. However, we can also see that a template function like `option` allows us to access the graph's option table in a simple way.

As for C code, our next task is to setup a key that, when used on the Ti*k*Z layer, will run our algorithm. For this, we can use an object derived from a `declarations`. In our example, the `FastLayout` is both derived from a `runner` (since it contains an algorithm) and also from `declarations` (since it also contains the code necessary for declaring this algorithm). If you prefer, you can split this into two classes. A `declarations` object must override the `declare` method. This method gets a `script` object as input, which is the "representation" of Lua inside the C++ code:

```
void declare(scripting::script s) {
  using namespace scripting;

  s.declare(key ("fast simple demo layout c++")
            .summary ("The C++ version of the hello world of graph drawing")
            .precondition ("connected")
            .precondition ("tree")
            .algorithm (this));

  s.declare(key ("fast simple demo radius c++")
            .summary ("A radius value for the hello world of graph drawing")
            .type ("length")
            .initial ("1cm"));
}
```

For each key that we wish to declare, we call the script's `declare` method once. This method takes a `key` object as input, which can be configured through a sequence of calls to different member functions (like `summary` or `algorithm`). Most of these member functions are rather self-explaining; only `algorithm` is a bit trickier: It does not take a function as input, but rather an object of type `runner` and it will call the `run` method of this object whenever the algorithm is run.

Lastly, we also need to write the entry point:

```
extern "C" int luaopen_pgf_gd_examples_c_SimpleDemoCPlusPlus (struct lua_State *state) {
  scripting::script s (state);
  s.declare (new FastLayout);
  return 0;
}
```

Note that it is the job of the interface classes to free the passed `declarations` object. For this reason, you really need to call `new` and cannot pass the address of a temporary object.

As before, because of the bug in some LuaTEX versions, to actually load the library at runtime, we need to rename it to

```
pgf_gd_examples_c_SimpleDemoCPlusPlus.so
```

and then say

```
require 'pgf_gd_examples_c_SimpleDemoCPlusPlus'
```

or in Ti*k*Z

```
\usegdlibrary {pgf_gd_examples_c_SimpleDemoCPlusPlus}
```

We can now use it:

```
\tikz \graph [fast simple demo layout c++, fast simple demo radius c++=1.25cm]
{ a -> b -> c -> d -> e -> a };
```

37.3.2 The Interface From C++

The header `graphdrawing/c/pgf/gd/interface/c/InterfaceFromC++.h` contains, as the name suggest, the interface from C++. A complete documentation is still missing, but let us go over the main ideas:

Runners. Algorithms are represented by objects of type `runner`. An algorithm will overwrite the `run` method, as we saw in the example, and it should modify the `parameters` of the runner object.

In addition to the `run` method, there are also two more virtual methods, called `bridge` and `unbrigde`. The first is called before the `run` method is called and the second afterwards. The idea is that another framework, such as OGDF, can implement a new class `ogdf_runner` that overrides these two methods in order to transform the Lua/C representation of the input graph into an OGDF representation prior to the `run` method being called. The `run` method can then access additional member variables that store the graph representations in OGDF form (or another form, depending on the framework). The `unbridge` method allows the framework to translate back.

Although a `runner` object must be created for every algorithm, an algorithm can also reside in a function. The class `function_runner` is a simple wrapper that turns a function into such an object.

Keys. A key object is a temporary object that is passed to the `declare` method of a script. It represents the table that is passed to the Lua function `declare`. In order to make setting its field easy, for each field name there is a corresponding function (like `summary`) that takes the string that should be set to this field and returns the key object once more, so that we can chain calls.

The `algorithm` method gets a runner object as parameter and will store a pointer to this object inside Lua. Each time the algorithm is used, this object will be used to "run" the algorithm, that is, the methods `prepare`, `bridge`, `run`, and `unbridge` will be called in that order. Since the object is reused each time, only one object is needed; but this object may not be freed prematurely. Indeed, you will normally create the object using `new` once and will then never delete it.

A typical idiom you may find in the code is

```
s.declare (key (...)
           .algorithm(this)
           ...);
```

This code is seen inside the `declare` method of objects that are both declarations and runners. They register "themselves" via the above code. Note, however, that this requires that the `this` pointer is not a temporary object. (The typing rules of C++ make it hard for this situation to happen, but it can be achieved.)

Reading options. Once options have been declared, your C++ algorithms will wish to read them back. For this, the `parameters` field of a runner object provides a number of templated methods:

- The `option_is_set` method returns `true` if the passed option has been set *and* can be cast to the type of the template. So, `option_is_set<double>("node distance")` will return true if the `node distance` key has been set for the graph as a whole (currently, there is no way to read the options of a vertex or an edge from C++, use the C methods instead).
- The `option` function comes in two flavours: First, it takes a single option name and just returns the option's value. If, however, the option has not been set or has another type, some sort of null value is returned. So, `option<double>("node distance")` will return the configured node distance as a double. When an option has an initial value, this call will always return a sensible value.

  The second flavour of `option` allows you to pass a reference to an object in which the option's value should be stored and the function will return true if the option is set (and, thus, something was written into the reference). This is the "safest" way to access an option:

  ```
  double dist;
  if (parameters->option ("node distance", dist))
    ...
  ```

  Caution must be taken for `char*` options: The returned string must be explicitly freed; it will be a copy of the string stored in the Lua table.
- The `configure_option` method is used to set a member of an object based on the value of a certain option. For this, you must pass a pointer to a member function that sets the member. Here is an example:

  ```
  class MyClass {
  public:
    void setMyDistance (double distance);
  ...
  };
  ...

  MyClass m;
  parameters->configure_option("node distance", &MyClass::setMyDistance, m);
  ```

  If the option has not been set or does not have the correct type, the member function is not called.

Factories and modules. A Lua key is normally either a Boolean, a double, or a string. However, in C++, we may also sometimes wish Lua users to configure which C function is used to achieve something. One could do this using strings or numbers and then use search algorithms or a long `switch`, but this would neither be flexible nor elegant.

Instead, it is possible to store *factories* in Lua keys. A factory is a class derived from `factory` that implements the virtual function `make`. This function will return a new object of a template type. You can store such a factory in a key.

The `make` method of a parameters object allows you to invoke the factory stored in a key. (If no factory is stored in it, `null` is returned).

The `configure_module` method is akin to `configure_option`, only the result of applying the factory is passed to the member function of the class.

Scripts. A "script" is the abstraction of the communication between Lua and C++. From C++'s point of view, the script object offers different `declare` methods that allow us to "make objects and function scriptable" in the sense that they can then be called and configured from Lua. The script must be initialized with a Lua state and will be bound to that state (basically, the script only stores this single pointer).

When you call `declare`, you either pass a single key object (which is then declared on the Lua layer) or you pass a `declarations` object, whose virtual `declare` method is then called. The `declarations` objects are used to bundle several declarations into a single one.

37.4 Writing Graph Drawing Algorithms Using OGDF

Built on top of the C++ interface, a small interface allows you to easily link algorithms written for the OGDF (Open Graph Drawing Framework) with graph drawing in Lua.

37.4.1 The Hello World of Graph Drawing in OGDF – From Scratch

We start with some startup code:

```
#include <pgf/gd/ogdf/c/InterfaceFromOGDF.h>
#include <math.h>

using namespace ogdf;
using namespace scripting;
```

Note that the interface from OGDF resides in the `ogdf` folder, not in the `interface` folder.

Like in the plain C++ interface, we must now subclass the `runner` class and the `declarations` class. Also like the plain C++ interface, we can use multiple inheritance. The difference lies in the fact that we do not directly subclass form `runner`, but rather from `ogdf_runner`. This class implements the complicated "bridging" or "translation" process between the world of `InterfaceFromC++` and OGDF:

```
struct FastLayoutOGDF : declarations, ogdf_runner {

  void run () {
    double angle  = 6.28318530718 / graph.numberOfNodes();
    double radius = parameters->option<double>("my radius ogdf");

    int i = 0;
    for (node v = graph.firstNode(); v; v=v->succ(), i++) {
      graph_attributes.x(v) = cos(angle*i) * radius;
      graph_attributes.y(v) = sin(angle*i) * radius;
    }
  }
```

As can be seen, in a subclass of `ogdf_runner`, the `run` method will have access to a member called `graph` and to another member called `graph_attributes`. These will have been setup with the graph from the Lua layer and, after the algorithm has run, the information stored in the `x` and `y` fields of the graph attributes and also the bend information of the edges will be written back automatically.

Next, we need to declare the algorithm. This is done as in the plain C++ interface:

```
  void declare(script s) {
    using namespace scripting;

    s.declare(key ("fast simple demo layout ogdf")
          .summary ("The OGDF version of the hello world of graph drawing")
          .precondition ("connected")
          .algorithm (this));

    s.declare(key ("my radius ogdf")
          .summary ("A radius value for the hello world of graph drawing")
          .type ("length")
          .initial ("1cm"));
  }
};
```

Finally, we need the entry point, which is also "as usual":

```
extern "C" int luaopen_pgf_gd_examples_c_SimpleDemoOGDF (struct lua_State *state) {
  script (state).declare (new FastLayoutOGDF);
  return 0;
}
```

Yet again, we need to rename the resulting shared library and then say `require` on it. We can now use it:

```
\tikz \graph [fast simple demo layout ogdf, my radius ogdf=1cm]
{ a -> b -> c -> d -> e -> a };
```

37.4.2 The Hello World of Graph Drawing in OGDF – Adapting Existing Classes

In the previous example we implemented a graph drawing algorithm using OGDF for use with Lua "from scratch". In particular, the whole algorithm was contained in the `run` method of our main class. In practice, however, graph drawing algorithms are typically placed in classes that "know nothing about scripting". For instance, our hello world of graph drawing might actually be implemented like this:

```
// File HelloWorldLayout.h
#include <ogdf/module/LayoutModule.h>

class HelloWorldLayout : public ogdf::LayoutModule {
public:

  virtual void call(ogdf::GraphAttributes &GA)
  {
    using namespace ogdf;

    const Graph &graph = GA.constGraph();
    double angle  = 6.28318530718 / graph.numberOfNodes();
    int i = 0;
    for (node v = graph.firstNode(); v; v=v->succ(), i++) {
      GA.x(v) = cos(angle*i) * radius;
      GA.y(v) = sin(angle*i) * radius;
    }
  }

  void setRadius (double r) { radius = r; }

private:

  double radius;
};
```

Now, what we actually want to do is to "make this class scriptable". For this, we setup a new class whose `run` method will produce a new `HelloWorldLayout`, configure it, and then run it. Here is this run method:

```
void run ()
{
  HelloWorldLayout layout;
  parameters->configure_option("HelloWorldLayout.radius", &HelloWorldLayout::setRadius, layout);
  layout.call(graph_attributes);
}
```

Next, we need to write the declarations code. This is very similar to the "from scratch" version:

```
void declare(script s) {
  using namespace scripting;

  s.declare(key ("HelloWorldLayout")
            .summary ("The OGDF version of the hello world of graph drawing")
            .precondition ("connected")
            .algorithm (this));

  s.declare(key ("HelloWorldLayout.radius")
            .summary ("A radius value for the hello world of graph drawing")
            .type ("length")
            .alias ("radius"));
}
```

Two remarks are in order: First, it is customary to name the keys for the display system the same way as the classes. Second, the different configuration options of the algorithm are named with the class name followed by the option name. This makes it clear who, exactly, is being configured. However, these keys should then also get an `alias` field set, which will cause an automatic forwarding of the key to something more "user friendly" like just `radius`.

It remains to put the above methods in a "script" file. It is this file that, when compiled, must be linked at runtime against LuaTEX

```
// File HelloWorldLayout_script.c++

#include <pgf/gd/ogdf/c/InterfaceFromOGDF.h>
#include <HelloWorldLayout.h>

using namespace ogdf;
using namespace scripting;

struct HelloWorldLayout_script : declarations, ogdf_runner {
  void run ()             { ... see above ... }
  void declare (script s) { ... see above ... }
};

extern "C" int luaopen_my_path_HelloWorldLayout_script (struct lua_State *state) {
  script (state).declare (new HelloWorldLayout_script);
  return 0;
}
```

37.4.3 Documenting OGDF Algorithms

As explained in Section 37.2.2, we can add external documentation to algorithms written in C and, using the `documentation_in` method of the `key` class, we can use the exact same method to document OGDF algorithms.

I strongly recommend making use of this feature since, currently, the documentation of many OGDF classes is sketchy at best and using Ti*k*Z examples seems to be a good way of explaining the effect of the different parameters algorithms offer.

37.4.4 The Interface From OGDF

The support for OGDF offered inside `InterfaceFromOGDF.h` is just the class `ogdf_runner` we saw already in the example. In addition, there is also a wrapper class `ogdf_function_runner` that allows you to wrap an algorithm implemented in a function that uses OGDF, but I expect this to be the case only rarely.

38 The Display Layer

by Till Tantau

> *You do not need to read this section in order to write new graph drawing algorithms. It is of interest only to those wishing to write programs that wish to "use" the graph drawing system in a similar way that Ti*k*Z does for laying out graphs that are generated and then passed down from the program to the graph drawing system.*

38.1 Introduction: The Interplay of the Different Layers

The job of the graph drawing system is to run graph drawing algorithms on graphs. Since graph drawing is useful in many different contexts, not only in the context of Ti*k*Z for which the present system was originally developed, the graph drawing system was designed in such a way that it can be used independently of Ti*k*Z. To achieve this, the *display layer* provides an interface via which an arbitrary program (Ti*k*Z, a graph editor, a command line interface, etc.) "talk" to the graph drawing system.

To better understand how this works, let us consider the following setup:

- A program would like to communicate with the graph drawing system. It is written in Java and its job is to analyse social networks. This software would like to use graph drawing to produce drawings of some "social graphs" arising from its analyses. This software will be called "the display system" in the following.
- There are two algorithms that the display system would like to apply to the graphs its produces. Let us call these algorithms "A" and "B". However, the display system would also, ideally, wish to make it possible that its user chooses other algorithms "C" that are not necessarily part of its binary.
- The display system has internally generated a "social graph" that it would now like to draw.

For this setup, the communication between the different layers of the graph drawing system is as follows:

1. The display system, being written in Java, must embed Lua to use the graph drawing system.
2. The display system must initialize the graph drawing system. For this, it must use `require` on the file `InterfaceToDisplay`, which, as the name suggests, contains the interface between the display system and the graph drawing system.

   It must also create a so-called "binding" between the graph drawing system and the display layer. See Section 39 for more information on bindings.
3. The display system now needs to load the declarations of the algorithms A and B. For this, it just needs to apply `require` to the files in which the algorithms reside. This will cause the `declare` function to be called. This function is declared by `InterfaceToAlgorithms` and allows files to declare that certain algorithms and options are now available.

   Once this is done, the graph drawing system is fully initialized and can be used.
4. Later on, the display system wishes to lay out a social graph. Note that this is known as "drawing the graph" in the graph drawing community, even though this only means the coordinates are computed for the nodes of the graph. The actual "rendering" of the graph on the display is the job of the display system (hence the name "display layer").
5. To start the layout process, the display system calls the function `beginGraphDrawingScope` of the class `InterfaceToDisplay`.
6. Next, for each vertex the function `createVertex` of the interface class must be called and similarly for each edge. These calls will cause an internal model of the graph to be created on the algorithm layer. Options that are attached to nodes and edges are also communicated to the graph drawing system through function calls like `pushOption`.
7. When the graph is fully specified, the method `runGraphDrawingAlgorithm` must be called, which is once more a method of the interface class. This function will internally discern which algorithms have been chosen (via options) and invoke the code of the appropriate algorithms.

8. Next, the function `renderGraph` must be called. Its job is to "communicate back" which coordinates have been computed. It will use the binding for this communication, see Section 39.

9. Finally, by calling `endGraphDrawingScope`, the resources needed for the layout process for the social graph are freed once more.

A few points may be noteworthy:

- The whole communication between the display system and the graph drawing system goes through the interface class via function calls. This makes it easy to communicate with display system whose internal model of graphs is quite different from the one used in Lua (as is certainly the case for TEX, but also the "social graph" mentioned above need not even exist as a separate entity inside the display system).

  The display system should only rely on the interface class. All communication has to go through this class, the display system may not access the internals of the internally constructed graphs directly.

- New algorithms can be loaded at runtime, as long as the `require` method is supported.

- The display system can also use the interface class to query which algorithms and which options are available in principle (and display this information to the user). The display system can even get access the documentation of the options at runtime since the documentation is stored in fields of the declared options.

In the following, we first present a simple display system other than Ti*k*Z. The remainder of the section then encompasses a documentation of the different functions of the class `InterfaceToDisplay`.

38.2 An Example Display System

In the following, we present the code of a very simple display system written in Lua (another such display system is Ti*k*Z itself, but the minimal system will allow us to focus on what is really needed). You will also find it in `pgf.gd.examples.ASCIIDisplayer`.

The job of this display system is to parse a string that encodes a graph and to call the appropriate functions of `InterfaceToDisplay` to lay out the graph. The actual calls for rendering the graph are part of the binding, which is documented in Section 39.4.

The syntax of the to-be-laid-out graph is a reduced version of Ti*k*Z's `graph` syntax: The string must start with `graph[`⟨*algorithm's name*⟩`]{` and end with `}`. Between the braces, all lines must either be of the form ⟨*node name*⟩`;` or ⟨*name 1*⟩`->`⟨*name 2*⟩`;` with optional spaces around the node names.

Let us now have a look at what we must do to use the graph drawing system. First, we load some libraries and initialize the binding (see Section 39.4 for details on the binding; we can ignore it for now).

```
local InterfaceToDisplay = require "pgf.gd.interface.InterfaceToDisplay"
InterfaceToDisplay.bind(require "pgf.gd.examples.BindingToASCII")

-- Load two libraries that define graph drawing algorithms. We can do this only *after* the binding
-- has been created since they call the declare function internally.
require "pgf.gd.layered.library"
require "pgf.gd.force.library"
```

Now comes some preparation code that starts a graph drawing scope and sets up the algorithm to the string provided between the square brackets:

```
local algorithm = io.read():match("%s*graph%s*%[(.-)%]")

InterfaceToDisplay.pushPhase(algorithm, "main", 1)
InterfaceToDisplay.pushOption("level distance", 6, 2)
InterfaceToDisplay.pushOption("sibling distance", 8, 3)
InterfaceToDisplay.beginGraphDrawingScope(3)
InterfaceToDisplay.pushLayout(4)
```

The numbers `1` to `4` are the positions on the options stack at which the options should be placed. See the description of `pushOption` for more details.

We are now ready to create the vertices and edges via a very simple parser:

```
for line in io.lines() do
  if line:match("}") then     break   elseif line:find("-") then
    local n1, dir, n2 = string.match(line, "^%s*(.-)%s*(-.)%s*(.-)%s*;")
    InterfaceToDisplay.createEdge(n1, n2, dir, 4)
  else
    local n1 = string.match(line, "^%s*(.-)%s*;")
    InterfaceToDisplay.createVertex(n1, "rectangle", nil, 4)
  end
end
```

The graph is now completely constructed inside the graph drawing system. We can now invoke the algorithms:

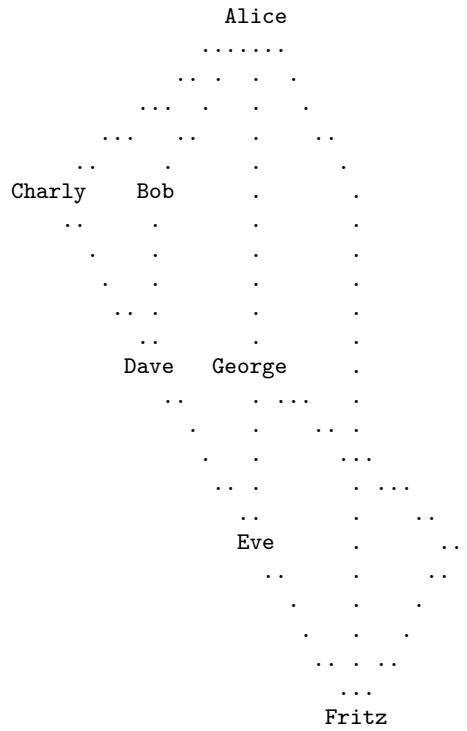
```
InterfaceToDisplay.runGraphDrawingAlgorithm()
InterfaceToDisplay.renderGraph()
InterfaceToDisplay.endGraphDrawingScope()
```

We can now run the resulting file using the Lua interpreter. If we provide the input shown on the left, we get the output shown on the right:

Input given to ASCIIDisplayer:

```
graph [layered layout] {
  Alice;
  Bob;
  Charly;
  Dave;
  Eve;
  Fritz;
  George;
  Alice -> Bob;
  Alice -> Charly;
  Charly -> Dave;
  Bob -> Dave;
  Dave -> Eve;
  Eve -> Fritz;
  Fritz -> Alice;
  George -> Eve;
  George -> Fritz;
  Alice -> George;
}
```

Output produced by ASCIIDisplayer:



38.3 The Interface to Display Systems

Lua table `InterfaceToDisplay` (declared in `pgf.gd.interface.InterfaceToDisplay`)

This class provides the interface between a display layer (like Ti*k*Z or a graph editor) and graph drawing system. Another class, `InterfaceToAlgorithms`, binds the algorithm layer (which are written in Lua) to the graph drawing system.

The functions declared here are independent of the actual display layer. Rather, the differences between the layers are encapsulated by subclasses of the `Binding` class, see that class for details. Thus, when a new display layer is written, the present class is *used*, but not *modified*. Instead, only a new binding is created and all display layer specific interaction is put there.

The job of this class is to provide convenient methods that can be called by the display layer. For instance, it provides methods for starting a graph drawing scope, managing the stack of such scope, adding a node to a graph and so on.

Alphabetical method summary:

`function InterfaceToDisplay.addToVertexOptions (name, height)`

```
function InterfaceToDisplay.beginGraphDrawingScope (height)
function InterfaceToDisplay.bind (class)
function InterfaceToDisplay.createEdge (tail, head, direction, height, binding_infos)
function InterfaceToDisplay.createEvent (kind, param)
function InterfaceToDisplay.createVertex (name, shape, path, height, binding_infos, anchors)
function InterfaceToDisplay.endGraphDrawingScope ()
function InterfaceToDisplay.getDeclaredKeys ()
function InterfaceToDisplay.pushLayout (height)
function InterfaceToDisplay.pushOption (key, value, height)
function InterfaceToDisplay.pushSubgraphVertex (name, height, info)
function InterfaceToDisplay.renderGraph ()
function InterfaceToDisplay.resumeGraphDrawingCoroutine ()
function InterfaceToDisplay.runGraphDrawingAlgorithm ()
```

`function InterfaceToDisplay.bind(class)`

Initialize the binding. This function is called once by the display layer at the very beginning. For instance, Ti*k*Z does the following call:

```
InterfaceToDisplay.bind(require "pgf.gd.bindings.BindingToPGF")
```

Inside this call, many standard declarations will be executed, that is, the declared binding will be used immediately.

Subsequently, the `binding` field of the `InterfaceCore` can be used.

Parameters: 1. `class` A subclass of `Binding`.

`function InterfaceToDisplay.beginGraphDrawingScope(height)`

Start a graph drawing scope. Note that this is not the same as starting a subgraph / sublayout, which are local to a graph drawing scope: When a new graph drawing scope is started, it is pushed on top of a stack of graph drawing scopes and all other "open" scopes are no longer directly accessible. All method calls to an `Interface...` object will refer to this newly created scope until either a new scope is opened or until the current scope is closed once more.

Each graph drawing scope comes with a syntactic digraph that is build using methods like `addVertex` or `addEdge`.

Parameters: 1. `height` The to-be-used height of the options stack. All options above this height will be popped prior to attacking the options to the syntactic digraph.

`function InterfaceToDisplay.runGraphDrawingAlgorithm()`

Arranges the current graph using the specified algorithm and options.

This function should be called after the graph drawing scope has been opened and the syntactic digraph has been completely specified. It will now start running the algorithm specified through the `algorithm_phase` options.

Internally, this function creates a coroutine that will run the current graph drawing algorithm. Coroutines are needed since a graph drawing algorithm may choose to create a new node. In this case, the algorithm needs to be suspended and control must be returned back to the display layer, so that the node can be typeset in order to determine the precise size information. Once this is done, control must be passed back to the exact point inside the algorithm where the node was created. Clearly, all of these actions are exactly what coroutines are for.

Returns: 1. Time it took to run the algorithm

`function InterfaceToDisplay.resumeGraphDrawingCoroutine()`

Resume the graph drawing coroutine.

This function is the work horse of the coroutine management. It gets called whenever control passes back from the display layer to the algorithm level. We resume the graph drawing coroutine so that the algorithm can start/proceed. The tricky part is when the algorithm yields, but is not done. In this case, the code needed for creating a new node is passed back to the display layer through the binding, which must then execute the code and then resuming the coroutine.

`function InterfaceToDisplay.endGraphDrawingScope()`

Ends the current graph drawing scope.

`function InterfaceToDisplay.createVertex(name, shape, path, height, binding_infos, anchors)`

Creates a new vertex in the syntactic graph of the current graph drawing scope. The display layer should call this function for each node of the graph. The `name` must be a unique string identifying the node. The newly created vertex will be added to the syntactic digraph. The binding function `everyVertexCreation` will then be called, allowing the binding to store information regarding the newly created vertex.

For each vertex an event will be created in the event sequence. This event will have the kind `"node"` and its `parameter` will be the vertex.

Parameters: *1.* `name` Name of the vertex.

2. `shape` The shape of the vertex such as `"circle"` or `"rectangle"`. This shape may help a graph drawing algorithm figuring out how the node should be placed.

3. `path` A `Path` object representing the vertex's path.

4. `height` The to-be-used height of the options stack. All options above this height will be popped prior to attacking the options to the syntactic digraph.

5. `binding_infos` These options are passed to and are specific to the current `Binding`.

6. `anchors` A table of anchors (mapping anchor positions to `Coordinates`).

`function InterfaceToDisplay.pushSubgraphVertex(name, height, info)`

Creates a new vertex in the syntactic graph of the current graph drawing scope that is a subgraph vertex. Such a vertex "surrounds" the vertices of a subgraph. The special property of a subgraph node opposed to a normal node is that it is created only after the subgraph has been laid out. However, the difference to a collection like `hyper` is that the node is available immediately as a normal node in the sense that you can connect edges to it.

What happens internally is that subgraph nodes get "registered" immediately both on the display level and on the algorithm level, but the actual node is only created inside the layout pipeline using a callback of the binding. The present function is used to perform this registering. The node creation happens when the innermost layout in which the subgraph node is declared has finished. For each subgraph node, a collection is created that contains all vertices (and edges) being part of the subgraph. For this reason, this method is a `push...` method, since it pushes something on the options stack.

The `init` parameter will be used during the creation of the node, see `Binding:createVertex` for details on the fields. Note that `init.text` is often not displayed for such "vast" nodes as those created for whole subgraphs, but a shape may use it nevertheless (for instance, one might display this text at the top of the node or, in case of a UML package, in a special box above the actual node).

The `init.generated_options` will be augmented by additional key–value pairs when the vertex is created:

- The key `subgraph point cloud` will have as its value a string that is be a list of points (without separating commas) like `"(10pt,20pt)(0pt,0pt)(30pt,40pt)"`, always in this syntax. The list will contain all points inside the subgraph. In particular, a bounding box around these points will encompass all nodes and bend points of the subgraph. The bounding box of this point cloud is guaranteed to be centered on the origin.
- The key `subgraph bounding box width` will have as its value the width of a bounding box (in TEX points, as a string with the suffix `"pt"`).
- The key `subgraph bounding box height` stores the height of a bounding box.

Parameters: *1.* `name` The name of the node. *2.* `height` Height of the options stack. Note that this method pushes something (namely a collection) on the options stack. *3.* `info` A table passed to `Binding:createVertex`, see that function.

`function InterfaceToDisplay.addToVertexOptions(name, height)`

Add options for an already existing vertex.

This function allows you to add options to an already existing vertex. The options that will be added are all options on the current options stack; they will overwrite existing options of the same name. For collections, the vertex stays in all collections it used to, it is only added to all collections that are currently on the options stack.

Parameters: 1. `name` Name of the vertex. *2.* `height` The option stack height.

`function InterfaceToDisplay.createEdge(tail, head, direction, height, binding_infos)`

Creates a new edge in the syntactic graph of the current graph drawing scope. The display layer should call this function for each edge that is created. Both the `from` vertex and the `to` vertex must exist (have been created through `createVertex`) prior to your being able to call this function.

After the edge has been created, the binding layer's function `everyEdgeCreation` will be called, allowing the binding layer to store information about the edge.

For each edge an event is created, whose kind is `"edge"` and whose `parameter` is a two-element array whose first entry is the edge's arc in the syntactic digraph and whose second entry is the position of the edge in the arc's array of syntactic edges.

Parameters: 1. `tail` Name of the node the edge begins at. *2.* `head` Name of the node the edge ends at. *3.* `direction` Direction of the edge (e.g. `--` for an undirected edge or `->` for a directed edge from the first to the second node). *4.* `height` The option stack height, see for instance `createVertex`.

5. `binding_infos` These options will be stored in the `storage` of the vertex at the field index by the binding.

`function InterfaceToDisplay.pushOption(key, value, height)`

Push an option to the stack of options.

As a graph is parsed, a stack of "current options" is created. To add something to this table, the display layers may call the method `pushOption`. To pop something from this stack, just set the `height` value during the next push to the position to which you actually wish to push something; everything above and including this position will be popped from the stack.

When an option is pushed, several additional options may also be pushed, namely whenever the option has a `use` field set. These additional options may, in turn, also push new options. Because of this, this function returns a new stack height, representing the resulting stack height.

In addition to this stack height, this function returns a Boolean value indicating whether a "main algorithm phase was set". This happens whenever a key is executed (directly or indirectly through the `use` field) that selects an algorithm for the "main" algorithm phase. This information may help the caller to setup the graph drawing scopes correctly.

Parameters: 1. `key` A parameter (must be a string). *2.* `value` A value (can be anything). If it is a string, it will be converted to whatever the key expects. *3.* `height` A stack height at which to insert the key. Everything above this height will be removed.

Returns: 1. A new stack height

Returns: 1. A Boolean that is `true` if the main algorithm phase was set by the option or one option `use`d by it.

Returns: 1. The newly created entry on the stack. If more entries are created through the use of the `use` field, the original entry is returned nevertheless.

`function InterfaceToDisplay.pushLayout(height)`

Push a layout on the stack of options. As long as this layout is on the stack, all vertices and edges will be part of this layout. For details on layouts, please see `Sublayouts`.

Parameters: 1. `height` A stack height at which to insert the key. Everything above this height will be removed.

`function InterfaceToDisplay.createEvent(kind, param)`

Creates an event and adds it to the event string of the current scope.

Parameters: 1. `kind` Name/kind of the event. *2.* `parameters` Parameters of the event.

Returns: 1. The newly pushed event

`function InterfaceToDisplay.getDeclaredKeys()`

This method allows you to query the table of all declared keys. It contains them both as an array and also as a table index by the keys's names. In particular, you can then iterate over it using `ipairs` and you can check whether a key is defined by accessing the table at the key's name. Each entry of the table is the original table passed to `InterfaceToAlgorithms.declare`.

Returns: *1.* A lookup table of all declared keys.

`function InterfaceToDisplay.renderGraph()`

Renders the graph.

This function is called after the graph has been laid out by the graph drawing algorithms. It will trigger a sequence of calls to the binding layer that will, via callbacks, start rendering the whole graph.

In detail, this function calls:

```
local binding = InterfaceCore.binding

binding:renderStart()
render_vertices()
render_edges()
render_collections()
binding:renderStop()
```

Here, the `render_...` functions are local, internal functions that are, nevertheless, documented here.

Parameters: *1.* `name` Returns the algorithm class that has been declared using `declare` under the given name.

`function render_vertices(vertices)`

Render the vertices after the graph drawing algorithm has finished. This function is local and internal and included only for documenting the call graph.

When the graph drawing algorithm is done, the interface will start rendering the vertices by calling appropriate callbacks of the binding layer.

Consider the following code:

```
\graph [... layout] {
  a -- b -- c -- d;
};
```

In this case, after the graph drawing algorithm has run, the present function will call:

```
local binding = InterfaceCore.binding

binding:renderVerticesStart()
binding:renderVertex(vertex_a)
binding:renderVertex(vertex_b)
binding:renderVertex(vertex_c)
binding:renderVertex(vertex_d)
binding:renderVerticesStop()
```

Parameters: *1.* `vertices` An array of all vertices in the syntactic digraph.

`function render_collections(collections)`

Render the collections whose layer is not `0`. This local, internal function is called to render the different collection kinds.

Collection kinds rendered in the order provided by the `layer` field passed to `declare` during the declaration of the collection kind, see also `declare_collection`. If several collection kinds have the same layer, they are rendered in lexicographical ordering (to ensure that they are always rendered in the same order).

Consider the following code:

```
declare { key = "hyper", layer = 1 }
```

you can say on the Ti*k*Z layer

```
\graph {
  a, b, c, d;
  { [hyper] a, b, c }
  { [hyper] b, c, d }
};
```

In this case, after the graph drawing algorithm has run, the present function will call:

```
local binding = InterfaceCore.binding

binding:renderCollectionStartKind("hyper", 1)
binding:renderCollection(collection_containing_abc)
binding:renderCollection(collection_containing_bcd)
binding:renderCollectionStopKind("hyper", 1)
```

Parameters: *1.* `collections` The `collections` table of the current scope.

`function render_edges(arcs)`

Render the syntactic edges of a graph after the graph drawing algorithm has finished. This function is local and internal and included only for documenting the call graph.

When the graph drawing algorithm is done, the interface will first rendering the vertices using `render_vertices`, followed by calling this function, which in turn calls appropriate callbacks to the binding layer.

Consider the following code:

```
\graph [... layout] {
  a -- b -- c -- d;
};
```

In this case, after the graph drawing algorithm has run, the present function will call:

```
local binding = InterfaceCore.binding

binding:renderEdgesStart()
binding:renderEdge(edge_from_a_to_b)
binding:renderEdge(edge_from_b_to_c)
binding:renderEdge(edge_from_c_to_d)
binding:renderEdgesStop()
```

Parameters: *1.* `arcs` The array of arcs of the syntactic digraph.

`function get_current_options_table(height, table)`

Get the current options table.

An option table can be accessed like a normal table; however, there is a global fallback for this table. If an index is not defined, the value of this index in the global fallback table is used. (This reduces the overall amount of option keys that need to be stored with object.)

(This function is local and internal and included only for documentation purposes.)

Parameters: *1.* `height` The stack height for which the option table is required. *2.* `table` If non `nil`, the options will be added to this table.

Returns: *1.* The option table as described above.

39 The Binding Layer

39.1 Overview

This section explains how the *binding* of the graph drawing system to a particular display layer works. Let me stress that all of this is important only for readers who

- either wish to write new display system (see Section 38)
- or wish to know more about how the graph drawing system works on the pure PGF layer (this is were the binding occurs).

Bindings are used to encapsulate the details of the communication between the graph drawing system and a display system (see Section 38 for an introduction to display systems).

Consider a display system that communicates with the graph drawing system. At some point, the display system would like to run an algorithm to lay out a graph. To achieve this, it will call different functions from the class `InterfaceToDisplay` and the effect of this is that a representation of the to-be-drawn graph is constructed internally and that the appropriate algorithms are run. All of this is in some sense independent of the actual display system, the class `InterfaceToDisplay` offers the same standard interface to all display systems.

At some point, however, the graph drawing system may need to "talk back" to the display system. For instance, once the graph has been laid out, to trigger the actual rendering of the graph, the graph drawing system must "tell" the display layer where the vertices lie. For some display systems this is easy: if the display system itself is written in Lua, it could just access the syntactic digraph directly. However, for systems like Ti*k*Z or systems written in another language, the graph drawing system needs a set of functions that it can call that will tell the display system what is going on. This is were bindings come in.

The class `Binding` is an interface that defines numerous methods that will be called by the graph drawing system in different situations (see the documentation below for details). For instance, there is a function `renderVertex` that is called by the graph drawing system whenever a vertex should be rendered by the display system. The class is really just an interface in the sense of object-oriented programming. For each display system you need to create a subclass of `Binding` like `BindingToPGF` or `BindingToASCII` that implement the methods declared by `Binding`. The number of methods that need to be implemented depends on the display system.

In the following, you will find the documentation of the `Binding` class in Section 39.2. Following this, we first have a quick look at how the `BindingToPGF` works and then go over a simple example of a binding to a more or less imaginary display system. This example should help readers interested in implementing their own bindings.

39.2 The Binding Class and the Interface Core

Lua table `Binding` (declared in `pgf.gd.bindings.Binding`)

This class provides a (non-functional) default implementation of a binding between a display layer and the algorithm layer. Subclasses must implement most of the member functions declared in this class.

A instance of a subclass of this class encapsulates the complete implementation of all specific code needed for the communication between the display layer and the graph drawing engine.

Note that you never call the methods of a `Binding` object directly, neither from the display layer nor from the algorithm layer. Instead, you use the more powerful and more easy to use functions from `InterfaceToDisplay` and `InterfaceToAlgorithms`. They call the appropriate `Binding` methods internally.

Because of the above, in order to use the graph drawing system inside a new display layer, you need to subclass `Binding` and implement all the functions. Then you need to write the display layer in such a way that it calls the appropriate functions from `InterfaceToDisplay`.

Field `storage`

A `Storage` storing the information passed from the display layer. The interpretation of this left to the actual binding.

Alphabetical method summary:

`function Binding:createVertex (init)`

```
function Binding:declareCallback (t)
function Binding:everyEdgeCreation (e)
function Binding:everyVertexCreation (v)
function Binding:renderCollectionStartKind (kind, layer)
function Binding:renderCollectionStopKind (kind, layer)
function Binding:renderCollection (collection)
function Binding:renderEdgesStart ()
function Binding:renderEdgesStop ()
function Binding:renderEdge (edge)
function Binding:renderStart ()
function Binding:renderStop ()
function Binding:renderVertex (vertex)
function Binding:renderVerticesStart ()
function Binding:renderVerticesStop ()
```

`function Binding:declareCallback(t)`

Declare a new key. This callback is called by `declare`. It is the job of the display layer to make the parameter `t.key` available to the parsing process. Furthermore, if `t.initial` is not `nil`, the display layer must convert it into a value that is stored as the initial value and call `InterfaceToDisplay.setOptionInitial`.

Parameters: 1. `t` See `InterfaceToAlgorithms.declare` for details.

`function Binding:renderStart()`

This function and, later on, `renderStop` are called whenever the rendering of a laid-out graph starts or stops. See `InterfaceToDisplay.render` for details.

`function Binding:renderStop()`

See `renderStart`.

`function Binding:renderCollectionStartKind(kind, layer)`

This function and the corresponding `...Stop...` functions are called whenever a collection kind should be rendered. See `InterfaceToDisplay.render_collections` for details.

Parameters: 1. `kind` The kind (a string). *2.* `layer` The kind's layer (a number).

`function Binding:renderCollectionStopKind(kind, layer)`

The counterpart to `renderCollectionStartKind`.

Parameters: 1. `kind` The kind. *2.* `layer` The kind's layer.

`function Binding:renderCollection(collection)`

Renders a single collection, see `renderCollectionStartKind` for details.

Parameters: 1. `collection` The collection object.

`function Binding:renderVerticesStart()`

This function and the corresponding `...Stop...` functions are called whenever a vertex should be rendered. See `InterfaceToDisplay.render_vertices` for details.

`function Binding:renderVerticesStop()`

The counterpart to `renderVerticesStop`.

`function Binding:renderVertex(vertex)`

Renders a single vertex, see `renderVertexStartKind` for details.

Parameters: 1. `vertex` The `Vertex` object.

`function Binding:everyVertexCreation(v)`

This method is called by the interface to the display layer after the display layer has called `createVertex` to create a new vertex. After having done its internal bookkeeping, the interface calls this function to allow the binding to perform further bookkeeping on the node. Typically, this will be done using the information stored in `Binding.infos`.

Parameters: 1. `v` The vertex.

`function Binding:renderEdgesStart()`

This function and the corresponding `...Stop...` functions are called whenever an edge should be rendered. See `InterfaceToDisplay.render_edges` for details.

`function Binding:renderEdgesStop()`

The counterpart to `renderEdgesStop`.

`function Binding:renderEdge(edge)`

Renders a single vertex, see `renderEdgeStartKind` for details.

Parameters: 1. `edge` The `Edge` object.

`function Binding:everyEdgeCreation(e)`

Like `everyVertexCreation`, only for edges.

Parameters: 1. `e` The edge.

`function Binding:createVertex(init)`

Generate a new vertex. This method will be called when the *algorithm* layer wishes to trigger the creation of a new vertex. This call will be made while an algorithm is running. It is now the job of the binding to cause the display layer to create the node. This is done by calling the `yield` method of the scope's coroutine.

Parameters: 1. `init` A table of initial values for the node. The following fields will be used:

- `name` If present, this name will be given to the node. If not present, an internal name is generated. Note that, unless the node is a subgraph node, this name may not be the name of an already present node of the graph; in this case an error results.
- `shape` If present, a shape of the node.
- `generated_options` A table that is passed back to the display layer as a list of key–value pairs.
- `text` The text of the node, to be passed back to the higher layer. This is what should be displayed as the node's text.

Lua table `InterfaceCore` (declared in `pgf.gd.interface.InterfaceCore`)

This class provides the core functionality of the interface between all the different layers (display layer, binding layer, and algorithm layer). The two classes `InterfaceToAlgorithms` and `InterfaceToDisplay` use, in particular, the data structures provided by this class.

Field `binding`

This field stores the "binding". The graph drawing system is "bound" to the display layer through such a binding (a subclass of `Binding`). Such a binding can be thought of as a "driver" in operating systems terminology: It is a small set of functions needed to adapt the functionality to one specific display system. Note that the whole graph drawing scope is bound to exactly one display layer; to use several bindings you need to setup a completely new Lua instance.

Field `scopes`

This is a stack of graph drawing scopes. All interface methods refer to the top of this stack.

Field `collection_kinds`

This table stores which collection kinds have been defined together with their properties.

Field `algorithm_classes`

A table that maps algorithm keys (like `tree layout` to class objects).

Field keys

A lookup table of all declared keys. Each entry of this table consists of the original entry passed to the `declare` method. Each of these tables is both index at a number (so you can iterate over it using `ipairs`) and also via the key's name.

Alphabetical method summary:

`function InterfaceCore.convert (s,t)`

`function InterfaceCore.topScope ()`

`function InterfaceCore.topScope()`

Returns the top scope

Returns: *1.* The current top scope, which is the scope in which everything should happen right now.

`function InterfaceCore.convert(s,t)`

Converts parameters types. This method is used by both the algorithm layer as well as the display layer to convert strings into the different types of parameters. When a parameter is pushed onto the option stack, you can either provide a value of the parameter's type; but you can also provide a string. This string can then be converted by this function to a value of the correct type.

Parameters: *1.* `s` A parameter value or a string. *2.* `t` The type of the parameter

Returns: *1.* If `s` is not a string, it is just returned. If it is a string, it is converted to the type `t`.

39.3 The Binding To PGF

Lua table BindingToPGF (declared in `pgf.gd.bindings.BindingToPGF`)

This class, which is a subclass of `Binding`, binds the graph drawing system to the PGF display system by overriding (that is, implementing) the methods of the `Binding` class. As a typical example, consider the implementation of the function `renderVertex`:

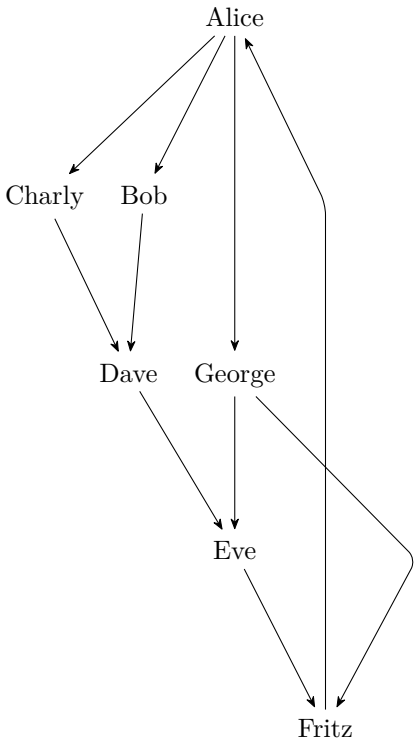
```
function BindingToPGF:renderVertex(v)
  local info = assert(self.infos[v], "thou shalt not modify the syntactic digraph")
  tex.print(
    string.format(
      "\\pgfgdcallbackrendernode{%s}{%fpt}{%fpt}{%fpt}{%fpt}{%fpt}{%fpt}{%s}",
      'not yet positionedPGFINTERNAL' .. v.name,
      info.x_min,
      info.x_max,
      info.y_min,
      info.y_max,
      v.pos.x,
      v.pos.y,
      info.box_count))
end
```

As can be seen, the main job of this function is to call a function on the TEX layer that is called `\pgfgdcallbackrendernode`, which gets several parameters like the name of the to-be-rendered node or the (new) position for the node. For almost all methods of the `Binding` class there is a corresponding "callback" macro on the TEX layer, all of which are implemented in the PGF library `graphdrawing`. For details on these callbacks, please consult the code of that file and of the class `BindingToPGF` (they are not documented here since they are local to the binding and should not be called by anyone other than the binding class).

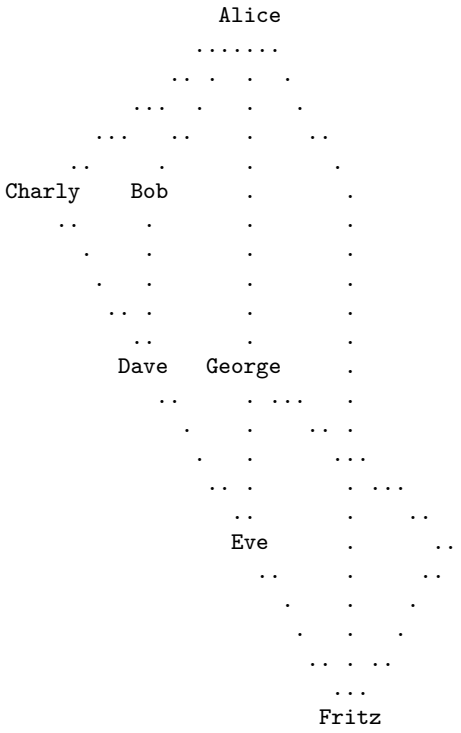
39.4 An Example Binding Class

In the present section a complete binding is presented to an imaginary "ASCII art display system" is presented. The idea is that this display system will depict graphs using just normal letters and spaces so that, when the text is typeset in a monospace font, a visualization of the graph results. For instance:

Graph rendered by ***BindingToPGF:***



Graph rendered by ***BindingToASCII:***



The binding will reside in a file `BindingToASCII.lua`, whose contents is detailed below, and which is used by calling the `bind` function of `InterfaceToDisplay`, see its documentation for details.

The binding's code starts with some initializations:

```
-- File BindingToASCII.lua

-- Imports
local lib = require "pgf.gd.lib"

-- Subclass the Binding class:
local BindingToASCII = lib.class { base_class = require "pgf.gd.bindings.Binding" }
```

The interesting code is the code for "rendering" a graph. The graph drawing system will invoke the binding's methods `renderStart` and `renderStop` to signal that the graph drawing algorithms have finished and that the vertices and edges can now be drawn.

In our ASCII renderer, we use a two-dimensional field holding characters that severs as the "drawing canvas". At the beginning of the rendering, we initialize it with blanks:

```
local canvas

function BindingToASCII:renderStart()
  canvas = {}
  -- Clear the canvas
  for x=-30,30 do
    canvas [x] = {}
    for y=-30,30 do
      canvas[x][y] = ' '
    end
  end
end
```

In order to "render" a vertex, the graph drawing system will call the `renderVertex` method. The binding of Ti*k*Z does a lot of complicated things in this method to retrieve the underlying node's box from internal table and to somehow reinstall the box in TEX's output stream; for our ASCII binding things are much simpler: We simply put the vertex's name at the canvas position corresponding to the vertex's `pos` coordinate. Note that this simple version of an ASCII renderer does not try to scale things; thus, array out of bounds might occur here.

```
function BindingToASCII:renderVertex(v)
  canvas [math.floor(v.pos.x)][math.floor(v.pos.y)] = v.name
end
```

The rendering of edges is a more complicated process. Given two vertices, we put dots at the canvas positions between them; provided there are no vertices (so edges are behind the nodes). Here is the essential part of the code (for the complete code, have a look at `pgf/gd/examples/BindingToASCII.lua`):

```
function BindingToASCII:renderEdge(e)
  local function connect (p,q)
    -- Connect the points p and q
    local x1, y1, x2, y2 = math.floor(p.x+0.5), math.floor(p.y+0.5), math.floor(q.x+0.5), math.floor(q.y+0.5)
    ...
    local delta_x = x2-x1
    local delta_y = y2-y1
    ...
      local slope = delta_y/delta_x
      for i=x1,x2 do
        local x,y = i, math.floor(y1 + (i-x1)*slope + 0.5)

        if canvas[x][y] == " " then
          canvas[x][y] = '.'
        end
      end
    ...
  end

  -- Iterate over all points on the path from tail to head:
  local p = e.tail.pos
  for i=1,#e.path do
    connect(p, e.tail.pos + e.path[i])
    p = e.tail.pos + e.path[i]
  end
  connect(p, e.head.pos)
end
```

The methods `renderVertex` and `renderEdge` will be called once for each vertex and edge of the to-be-rendered graph. At the end, the `renderStop` method is called. In our case, this method will output the canvas using `print`. A slight complication arises when node names are longer than just one character. In this case, the following code "centers" them on their coordinate and makes sure that they do not get overwritten by the dots forming edges:

```
function BindingToASCII:renderStop()
  for y=10,-30,-1 do
    local t = {}
    for x=-30,30 do
      local s = canvas[x][y]
      for i=1,#s do
        pos = x+30+i-math.floor(#s/2)
        if not t[pos] or t[pos] == " " or t[pos] == "." then
          t[pos] = string.sub(s,i,i)
        end
      end
    end
    print(table.concat(t))
  end
end
```

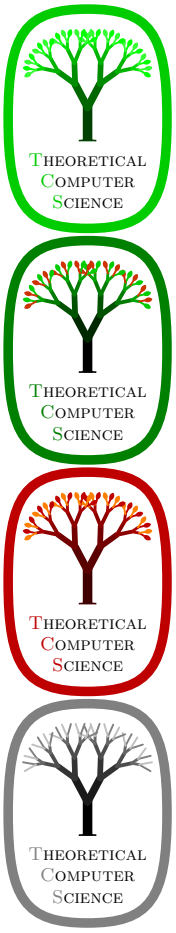
At the end, we need to return the created object:

```
return BindingToASCII
```

Part V
Libraries

by Till Tantau

In this part the library packages are documented. They provide additional predefined graphic objects like new arrow heads or new plot marks, but sometimes also extensions of the basic PGF or TikZ system. The libraries are not loaded by default since many users will not need them.




```
\usetikzlibrary {arrows,trees}
\tikzset{
  ld/.style={level distance=#1},lw/.style={line width=#1},
  level 1/.style={ld=4.5mm, trunk,          lw=1ex ,sibling angle=60},
  level 2/.style={ld=3.5mm, trunk!80!leaf a,lw=.8ex,sibling angle=56},
  level 3/.style={ld=2.75mm,trunk!60!leaf a,lw=.6ex,sibling angle=52},
  level 4/.style={ld=2mm,   trunk!40!leaf a,lw=.4ex,sibling angle=48},
  level 5/.style={ld=1mm,   trunk!20!leaf a,lw=.3ex,sibling angle=44},
  level 6/.style={ld=1.75mm,leaf a,         lw=.2ex,sibling angle=40},
}
\pgfarrowsdeclare{leaf}{leaf}
  {\pgfarrowsleftextend{-2pt} \pgfarrowsrightextend{1pt}}
{
  \pgfpathmoveto{\pgfpoint{-2pt}{0pt}}
  \pgfpatharc{150}{30}{1.8pt}
  \pgfpatharc{-30}{-150}{1.8pt}
  \pgfusepathqfill
}

\newcommand{\logo}[5]
{
  \colorlet{border}{#1}
  \colorlet{trunk}{#2}
  \colorlet{leaf a}{#3}
  \colorlet{leaf b}{#4}
  \begin{tikzpicture}
    \scriptsize\scshape
    \draw[border,line width=1ex,yshift=.3cm,
          yscale=1.45,xscale=1.05,looseness=1.42]
      (1,0) to [out=90, in=0]    (0,1)  to [out=180,in=90]  (-1,0)
            to [out=-90,in=-180] (0,-1) to [out=0,  in=-90] (1,0) -- cycle;

    \coordinate (root) [grow cyclic,rotate=90]
    child {
      child [line cap=round] foreach \a in {0,1} {
        child foreach \b in {0,1} {
          child foreach \c in {0,1} {
            child foreach \d in {0,1} {
              child foreach \leafcolor in {leaf a,leaf b}
                { edge from parent [color=\leafcolor,-#5] }
        } } }
      } edge from parent [shorten >=-1pt,serif cm-,line cap=butt]
    };

    \node [align=center,below] at (0pt,-.5ex)
    { \textcolor{border}{T}heoretical \\ \textcolor{border}{C}omputer \\
      \textcolor{border}{S}cience };
  \end{tikzpicture}
}
\begin{minipage}{3cm}
  \logo{green!80!black}{green!25!black}{green}{green!80}{leaf}\\
  \logo{green!50!black}{black}{green!80!black}{red!80!green}{leaf}\\
  \logo{red!75!black}{red!25!black}{red!75!black}{orange}{leaf}\\
  \logo{black!50}{black}{black!50}{black!25}{}
\end{minipage}
```

40 Three Dimensional Drawing Library

TikZ Library 3d

```
\usetikzlibrary{3d} % LaTeX and plain TeX
\usetikzlibrary[3d] % ConTeXt
```

This package provides some styles and options for drawing three dimensional shapes.

40.1 Coordinate Systems

Coordinate system xyz cylindrical

The `xyz cylindrical` coordinate system allows to you specify a point in terms of cylindrical coordinates, sometimes also referred to as cylindrical polar coordinates or polar cylindrical coordinates. It is very similar to the `canvas polar` and `xy polar` coordinate systems with the difference that you provide an elevation over the xy-plane using the `z` key.

/tikz/cs/angle=⟨*degrees*⟩ (no default, initially 0)

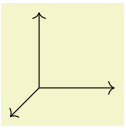
The angle of the coordinate interpreted in the ellipse whose axes are the x-vector and the y-vector.

/tikz/cs/radius=⟨*number*⟩ (no default, initially 0)

A factor by which the x-vector and y-vector are multiplied prior to forming the ellipse.

/tikz/cs/z=⟨*number*⟩ (no default, initially 0)

Factor by which the z-vector is multiplied.



```
\usetikzlibrary {3d}
\begin{tikzpicture}[->]
  \draw (0,0,0) -- (xyz cylindrical cs:radius=1);
  \draw (0,0,0) -- (xyz cylindrical cs:radius=1,angle=90);
  \draw (0,0,0) -- (xyz cylindrical cs:z=1);
\end{tikzpicture}
```

Coordinate system xyz spherical

The `xyz spherical` coordinate system allows you to specify a point in terms of spherical coordinates.

/tikz/cs/radius=⟨*number*⟩ (no default, initially 0)

Factor by which the x-, y-, and z-vector are multiplied.

/tikz/cs/latitude=⟨*degrees*⟩ (no default, initially 0)

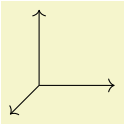
Angle of the coordinate between the y- and z-vector, measured from the y-vector.

/tikz/cs/longitude=⟨*degrees*⟩ (no default, initially 0)

Angle of the coordinate between the x- and y-vector, measured from the y-vector.

/tikz/cs/angle=⟨*degrees*⟩ (no default, initially 0)

Same as `longitude`.



```
\usetikzlibrary {3d}
\begin{tikzpicture}[->]
  \draw (0,0,0) -- (xyz spherical cs:radius=1);
  \draw (0,0,0) -- (xyz spherical cs:radius=1,latitude=90);
  \draw (0,0,0) -- (xyz spherical cs:radius=1,longitude=90);
\end{tikzpicture}
```

40.2 Coordinate Planes

Sometimes drawing with full three dimensional coordinates is not necessary and it suffices to draw in two dimensions but in a different coordinate plane. The following options help you to switch to a different plane.

40.2.1 Switching to an arbitrary plane

`/tikz/plane origin=`⟨*point*⟩ (no default, initially `(0,0)`)

Origin of the plane.

`/tikz/plane x=`⟨*point*⟩ (no default, initially `(1,0)`)

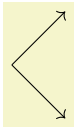
Unit vector of the x-direction in the new plane.

`/tikz/plane y=`⟨*point*⟩ (no default, initially `(0,1)`)

Unit vector of the y-direction in the new plane.

`/tikz/canvas is plane` (no value)

Perform the transformation into the new canvas plane using the units above. Note that you have to set the units *before* calling `canvas is plane`.



```
\usetikzlibrary {3d}
\begin{tikzpicture}[
    ->,
    plane x={(0.707,-0.707)},
    plane y={(0.707,0.707)},
    canvas is plane,
]
    \draw (0,0) -- (1,0);
    \draw (0,0) -- (0,1);
\end{tikzpicture}
```

40.2.2 Predefined planes

`/tikz/canvas is xy plane at z=`⟨*dimension*⟩ (no default)

A plane with

- `plane origin={(0,0,`⟨*dimension*⟩`)}`,
- `plane x={(1,0,`⟨*dimension*⟩`)}`, and
- `plane y={(0,1,`⟨*dimension*⟩`)}`.

`/tikz/canvas is yx plane at z=`⟨*dimension*⟩ (no default)

A plane with

- `plane origin={(0,0,`⟨*dimension*⟩`)}`,
- `plane x={(0,1,`⟨*dimension*⟩`)}`, and
- `plane y={(1,0,`⟨*dimension*⟩`)}`.

`/tikz/canvas is xz plane at y=`⟨*dimension*⟩ (no default)

A plane with

- `plane origin={(0,`⟨*dimension*⟩`,0)}`,
- `plane x={(1,`⟨*dimension*⟩`,0)}`, and
- `plane y={(0,`⟨*dimension*⟩`,1)}`.

`/tikz/canvas is zx plane at y=`⟨*dimension*⟩ (no default)

A plane with

- `plane origin={(0,`⟨*dimension*⟩`,0)}`,
- `plane x={(0,`⟨*dimension*⟩`,1)}`, and
- `plane y={(1,`⟨*dimension*⟩`,0)}`.

`/tikz/canvas is yz plane at x=`⟨*dimension*⟩ (no default)

A plane with

- `plane origin={(`⟨*dimension*⟩`,0,0)}`,

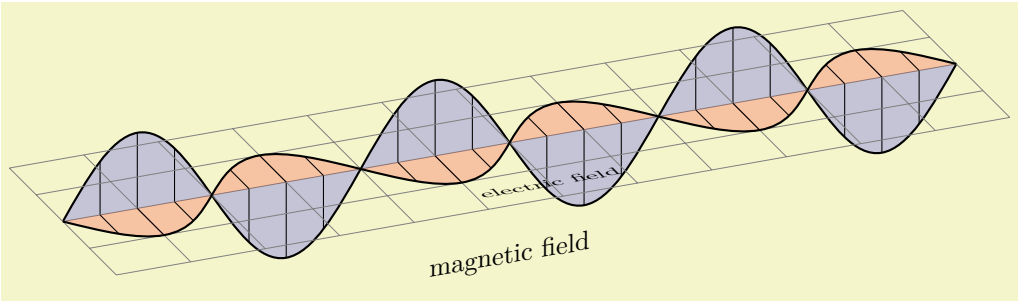
- `plane x={(`⟨*dimension*⟩`,1,0)}`, and
- `plane y={(`⟨*dimension*⟩`,0,1)}`.

`/tikz/canvas is zy plane at x=`⟨*dimension*⟩ (no default)

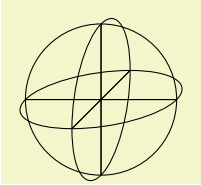
A plane with

- `plane origin={(`⟨*dimension*⟩`,0,0)}`,
- `plane x={(`⟨*dimension*⟩`,0,1)}`, and
- `plane y={(`⟨*dimension*⟩`,1,0)}`.

40.3 Examples




```
\usetikzlibrary {3d}
\begin{tikzpicture}[z={(10:10mm)},x={(-45:5mm)}]
  \def\wave{
    \draw[fill,thick,fill opacity=.2]
     (0,0) sin (1,1) cos (2,0) sin (3,-1) cos (4,0)
           sin (5,1) cos (6,0) sin (7,-1) cos (8,0)
           sin (9,1) cos (10,0)sin (11,-1)cos (12,0);
    \foreach \shift in {0,4,8}
    {
      \begin{scope}[xshift=\shift cm,thin]
        \draw (.5,0)  -- (0.5,0 |- 45:1cm);
        \draw (1,0)   -- (1,1);
        \draw (1.5,0) -- (1.5,0 |- 45:1cm);
        \draw (2.5,0) -- (2.5,0 |- -45:1cm);
        \draw (3,0)   -- (3,-1);
        \draw (3.5,0) -- (3.5,0 |- -45:1cm);
      \end{scope}
    }
  }
  \begin{scope}[canvas is zy plane at x=0,fill=blue]
    \wave
    \node at (6,-1.5) [transform shape] {magnetic field};
  \end{scope}
  \begin{scope}[canvas is zx plane at y=0,fill=red]
    \draw[help lines] (0,-2) grid (12,2);
    \wave
    \node at (6,1.5) [rotate=180,xscale=-1,transform shape] {electric field};
  \end{scope}
\end{tikzpicture}
```



```
\usetikzlibrary {3d}
\begin{tikzpicture}
  \begin{scope}[canvas is zy plane at x=0]
    \draw (0,0) circle [radius=1cm];
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}

  \begin{scope}[canvas is zx plane at y=0]
    \draw (0,0) circle [radius=1cm];
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}

  \begin{scope}[canvas is xy plane at z=0]
    \draw (0,0) circle [radius=1cm];
    \draw (-1,0) -- (1,0) (0,-1) -- (0,1);
  \end{scope}
\end{tikzpicture}
```

41 Angle Library

TikZ Library angles

```
\usetikzlibrary{angles} % LaTeX and plain TeX
\usetikzlibrary[angles] % ConTeXt
```

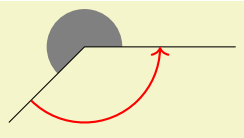
This library defines pic types for drawing angles.

Pic type angle=⟨*A*⟩--⟨*B*⟩--⟨*C*⟩

This pic adds a drawing of an angle to the current path. This "drawing of an angle" consist of a "sector" or "wedge" or "slice" whose pointed end is at point ⟨*B*⟩ and whose straight sides lie on the lines form ⟨*B*⟩ to ⟨*A*⟩ and from ⟨*B*⟩ to ⟨*C*⟩. The length of these lines is governed by the following key:

/tikz/angle radius=⟨*dimension*⟩ (no default, initially 5mm)

The length of the sides of the angle's wedge:

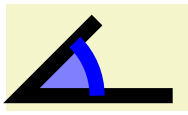


```
\usetikzlibrary {angles}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (-1,-1) coordinate (C)
            pic [fill=black!50]                      {angle = A--B--C}
            pic [draw,->,red,thick,angle radius=1cm] {angle = C--B--A};
```

The three points ⟨*A*⟩, ⟨*B*⟩, and ⟨*C*⟩ *must* be the names of nodes or coordinates; you cannot use direct coordinates like "`(1,1)`" here.

You can leave out the three points, in this case the text `A--B--C` is used; so in the above examples we could just have written `{angle}` in the first pic.

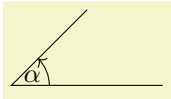
Concerning the sector that makes up the drawing of the angle, the angular part of this sector is drawn in front of the path if the `draw` option is given to the `pic`, while filled sector is drawn behind the `pic`, provided an option like `fill` or `shade` is passed to the pic. The following example shows the difference:



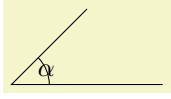
```
\usetikzlibrary {angles}
\tikz \draw [line width=2mm]
      (2,0) coordinate (A) -- (0,0) coordinate (B)
   -- (1,1) coordinate (C)
     pic [draw=blue, fill=blue!50, angle radius=1cm] {angle};
```

When `pic text` is set (which you typically do by using the quotes syntax), a node will be created whose name is empty (and, thus, inherits the pic's name) and which will be at the half-way angle between the lines to ⟨*A*⟩ and ⟨*C*⟩ and whose distance from ⟨*B*⟩ is `angle radius` times the following factor:

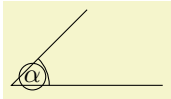
/tikz/angle eccentricity=⟨*factor*⟩ (no default, initially 0.6)




```
\usetikzlibrary {angles,quotes}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (1,1) coordinate (C)
  pic ["$\alpha$", draw, ->] {angle};
```




```
\usetikzlibrary {angles,quotes}
\tikz \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
          -- (1,1) coordinate (C)
  pic ["$\alpha$", draw, angle eccentricity=1] {angle};
```

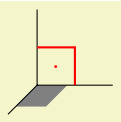



```
\usetikzlibrary {angles,quotes}
\tikz {
  \draw (2,0) coordinate (A) -- (0,0) coordinate (B)
     -- (1,1) coordinate (C)
      pic (alpha) ["$\alpha$", draw] {angle};

  \draw (alpha) circle [radius=5pt];
}
```

Pic type `right angle=⟨A⟩--⟨B⟩--⟨C⟩`

This pic adds a drawing of a right angle to the current path. It works in the same way as `angle` pic.



```
\usetikzlibrary {angles}
  \tikz
    \draw (0,0,0) coordinate (O)
      (1,0,0) coordinate (A) -- (O)
      (0,0,1) coordinate (B) -- (O)
      (0,1,0) coordinate (C) -- (O)
      pic [fill=gray,angle radius=4mm] {right angle = A--O--B}
      pic [draw,red,thick,angle eccentricity=.5,pic text=.]
        {right angle = A--O--C};
```

42 Arrow Tip Library

The libraries `arrows` and `arrows.spaced` from older versions of PGF are still available for compatibility, but they are considered deprecated.

The standard arrow tips, which are loaded by the library `arrows.meta`, are documented in Section 16.5.

43 Automata Drawing Library

TikZ Library automata

```
\usetikzlibrary{automata} % LaTeX and plain TeX
\usetikzlibrary[automata] % ConTeXt
```

This packages provides shapes and styles for drawing finite state automata and Turing machines.

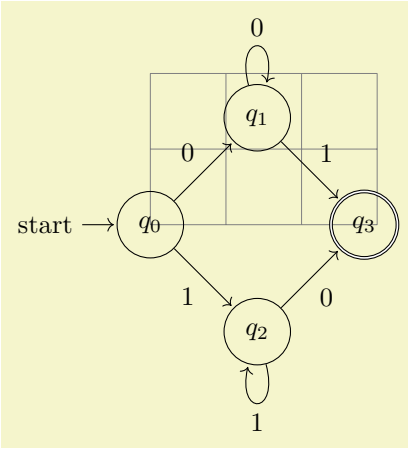
43.1 Drawing Automata

The `automata` (drawing) library is intended to make it easy to draw finite automata and Turing machines. It does not cover every situation imaginable, but most finite automata and Turing machines found in text books can be drawn in a nice and convenient fashion using this library.

To draw an automaton, proceed as follows:

1. For each state of the automaton, there should be one node with the option `state`.
2. To place the states, you can either use absolute positions or relative positions, using options like `above` or `right`.
3. Give a unique name to each state node.
4. Accepting and initial states are indicated by adding the options `accepting` and `initial`, respectively, to the state nodes.
5. Once the states are fixed, the edges can be added. For this, the `edge` operation is most useful. It is, however, also possible to add edges after each node has been placed.
6. For loops, use the `edge [loop]` operation.

Let us now see how this works for a real example. Let us consider a nondeterministic four state automaton that checks whether an input contains the sequence 0^*1 or the sequence 1^*0.




```
\usetikzlibrary {automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,auto]
  \draw[help lines] (0,0) grid (3,2);

  \node[state,initial]   (q_0)                    {$q_0$};
  \node[state]           (q_1) [above right=of q_0] {$q_1$};
  \node[state]           (q_2) [below right=of q_0] {$q_2$};
  \node[state,accepting](q_3) [below right=of q_1] {$q_3$};

  \path[->] (q_0) edge              node        {0} (q_1)
                  edge              node [swap] {1} (q_2)
            (q_1) edge              node        {1} (q_3)
                  edge [loop above] node        {0} ()
            (q_2) edge              node [swap] {0} (q_3)
                  edge [loop below] node        {1} ();
\end{tikzpicture}
```

43.2 States With and Without Output

The `state` style actually just "selects" a default underlying style. Thus, you can define multiple new complicated state style and then simply set the `state` style to your given style to get the desired kind of styles.

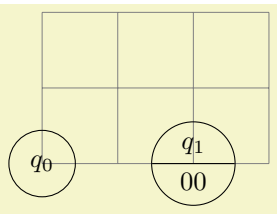
By default, the following state styles are defined:

`/tikz/state without output` (style, no value)

This node style causes nodes to be drawn as circles. Also, this style calls `every state`.

`/tikz/state with output` (style, no value)

This node style causes nodes to be drawn as split circles, that is, using the `circle split` shape. In the upper part of the shape you have the name of the style, in the lower part the output is placed. To specify the output, use the command `\nodepart{lower}` inside the node. This style also calls `every state`.



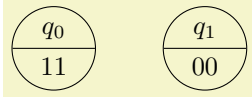

```
\usetikzlibrary {automata}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \node[state without output] {$q_0$};

  \node[state with output] at (2,0) {$q_1$ \nodepart{lower} $00$};
\end{tikzpicture}
```

`/tikz/state` (style, initially `state without output`)

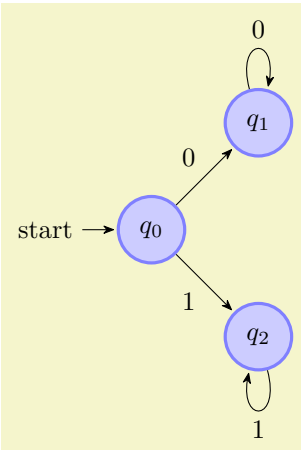
You should redefine it to something else, if you wish to use states of a different nature.




```
\usetikzlibrary {automata}
\begin{tikzpicture}[state/.style=state with output]
  \node[state]          {$q_0$ \nodepart{lower} $11$};
  \node[state] at (2,0) {$q_1$ \nodepart{lower} $00$};
\end{tikzpicture}
```

`/tikz/every state` (style, initially empty)

This style is used by `state with output` and also by `state without output`. By default, it does nothing, but you can use it to make your state look more fancy:




```
\usetikzlibrary {arrows.meta,automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid,>={Stealth[round]},
    every state/.style={draw=blue!50,very thick,fill=blue!20}]

  \node[state,initial]  (q_0)                      {$q_0$};
  \node[state]          (q_1) [above right=of q_0] {$q_1$};
  \node[state]          (q_2) [below right=of q_0] {$q_2$};

  \path[->] (q_0) edge              node [above left]  {0} (q_1)
                  edge              node [below left]  {1} (q_2)
            (q_1) edge [loop above] node               {0} ()
            (q_2) edge [loop below] node               {1} ();
\end{tikzpicture}
```

43.3 Initial and Accepting States

The styles `initial` and `accepting` are similar to the `state` style as they also just select an "underlying" style, which installs the actual settings for initial and accepting states.

Let us start with the initial states.

/tikz/initial (style, initially initial by arrow)

This style is used to draw initial states.

/tikz/initial by arrow (style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points from the text to the node. The node text and the direction and the distance can be set using the following key:

/tikz/initial text=⟨*text*⟩ (no default, initially start)

This key sets the text to be used. Use an empty text to suppress all text.

/tikz/initial where=⟨*direction*⟩ (no default, initially left)

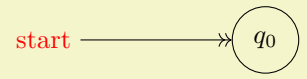
Set the place where the text should be shown. Allowed values are above, below, left, and right.

/tikz/initial distance=⟨*distance*⟩ (no default, initially 3ex)

Sets the length of the arrow leading from the text to the state node.

/tikz/every initial by arrow (style, initially empty)

This style is executed at the beginning of every path that contains the arrow and the text. You can use it to, say, make the text red or whatever.




```
\usetikzlibrary {automata}
\begin{tikzpicture}[every initial by arrow/.style={text=red,->>}]
  \node[state,initial,initial distance=2cm] {$q_0$};
\end{tikzpicture}
```

/tikz/initial above (style, no value)

This is a shorthand for initial by arrow,initial where=above.

/tikz/initial below (style, no value)

Works similarly to the previous option.

/tikz/initial left (style, no value)

Works similarly to the previous option.

/tikz/initial right (style, no value)

Works similarly to the previous option.

/tikz/initial by diamond (style, no value)

This style uses a diamond to indicate an initial node.

For the accepting states, the situation is similar: There is also an accepting style that selects the way accepting states are rendered. There are now two options: First, accepting by arrow, which works the same way as initial by arrow, only with the direction of arrow reversed, and accepting by double, where accepting states get a double line around them.

/tikz/accepting (style, initially accepting by double)

This style is used to draw accepting states. You can replace this by the style accepting by arrow to get accepting states with an arrow leaving them.

/tikz/accepting by double (style, no value)

This style causes a double line to be drawn around a state.

/tikz/accepting by arrow (style, no value)

This style causes an arrow and, possibly, some text to be added to the node. The arrow points to the text from the node.

The same options as for initial states can be used, only with initial replaced by accepting:

/tikz/accepting text=⟨*text*⟩ (no default, initially empty)

This key sets the text to be used.

`/tikz/accepting where=`⟨*direction*⟩ (no default, initially `right`)

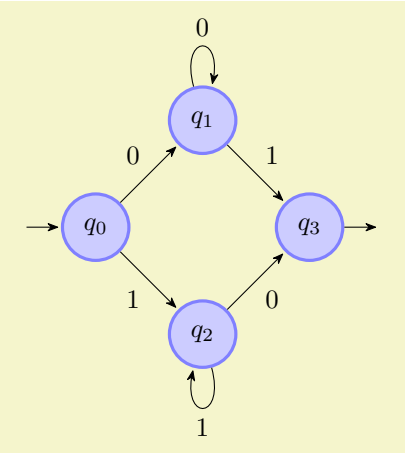
Set the place where the text should be shown. Allowed values are `above`, `below`, `left`, and `right`.

`/tikz/initial distance=`⟨*distance*⟩ (no default, initially `3ex`)

Sets the length of the arrow leading from the text to the state node.

`/tikz/every accepting by arrow` (style, initially empty)

Executed at the beginning of every path that contains the arrow and the text.




```
\usetikzlibrary {arrows.meta,automata,positioning}
\begin{tikzpicture}
  [shorten >=1pt,node distance=2cm,on grid,>={Stealth[round]},initial text=,
   every state/.style={draw=blue!50,very thick,fill=blue!20},
   accepting/.style=accepting by arrow]

  \node[state,initial]   (q_0)                      {$q_0$};
  \node[state]           (q_1) [above right=of q_0] {$q_1$};
  \node[state]           (q_2) [below right=of q_0] {$q_2$};
  \node[state,accepting](q_3) [below right=of q_1] {$q_3$};

  \path[->] (q_0) edge              node [above left]  {0} (q_1)
                  edge              node [below left]  {1} (q_2)
            (q_1) edge              node [above right] {1} (q_3)
                  edge [loop above] node               {0} ()
            (q_2) edge              node [below right] {0} (q_3)
                  edge [loop below] node               {1} ();
\end{tikzpicture}
```

`/tikz/accepting above` (style, no value)

This is a shorthand for `accepting by arrow,accepting where=above`.

`/tikz/accepting below` (style, no value)

Works similarly to the previous option.

`/tikz/accepting left` (style, no value)

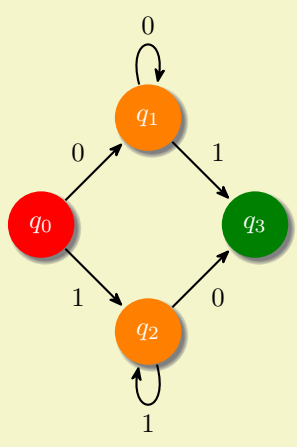
Works similarly to the previous option.

`/tikz/accepting right` (style, no value)

Works similarly to the previous option.

43.4 Examples

In the following example, we once more typeset the automaton presented in the previous sections. This time, we use the following rule for accepting/initial state: Initial states are red, accepting states are green, and normal states are orange. Then, we must find a path from a red state to a green state.

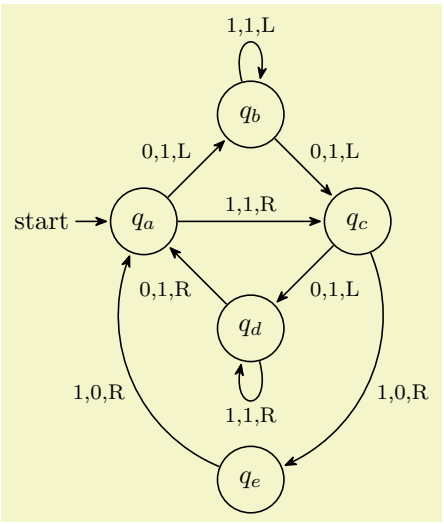



```
\usetikzlibrary {arrows.meta,automata,positioning,shadows}
\begin{tikzpicture}
  [shorten >=1pt,node distance=2cm,on grid,>={Stealth[round]},thick,
   every state/.style={fill,draw=none,orange,text=white,circular drop shadow},
   accepting/.style  ={green!50!black,text=white},
   initial/.style    ={red,text=white}]

  \node[state,initial]   (q_0)                      {$q_0$};
  \node[state]           (q_1) [above right=of q_0] {$q_1$};
  \node[state]           (q_2) [below right=of q_0] {$q_2$};
  \node[state,accepting](q_3) [below right=of q_1] {$q_3$};

  \path[->] (q_0) edge              node [above left]  {0} (q_1)
                  edge              node [below left]  {1} (q_2)
            (q_1) edge              node [above right] {1} (q_3)
                  edge [loop above] node               {0} ()
            (q_2) edge              node [below right] {0} (q_3)
                  edge [loop below] node               {1} ();
\end{tikzpicture}
```

The next example is the current candidate for the five-state busiest beaver:




```
\usetikzlibrary {arrows.meta,automata,positioning}
\begin{tikzpicture}[->,>={Stealth[round]},shorten >=1pt,
                    auto,node distance=2cm,on grid,semithick,
                    inner sep=2pt,bend angle=45]
  \node[initial,state] (A)                    {$q_a$};
  \node[state]         (B) [above right=of A] {$q_b$};
  \node[state]         (D) [below right=of A] {$q_d$};
  \node[state]         (C) [below right=of B] {$q_c$};
  \node[state]         (E) [below=of D]       {$q_e$};

  \path [every node/.style={font=\footnotesize}]
        (A) edge              node {0,1,L} (B)
            edge              node {1,1,R} (C)
        (B) edge [loop above] node {1,1,L} (B)
            edge              node {0,1,L} (C)
        (C) edge              node {0,1,L} (D)
            edge [bend left]  node {1,0,R} (E)
        (D) edge [loop below] node {1,1,R} (D)
            edge              node {0,1,R} (A)
        (E) edge [bend left]  node {1,0,R} (A);
\end{tikzpicture}
```

44 Babel Library

Ti*k*Z Library babel

```
\usetikzlibrary{babel} % LaTeX and plain TeX
\usetikzlibrary[babel] % ConTeXt
```

A tiny library that make the interaction with the `babel` package easier. Despite the name, it may also be useful in other contexts, namely whenever the catcodes of important symbols are changed globally. Normally, using this library is always a good idea; it is not always loaded by default since in some rare cases it may break old code.

The problems this library tries to fix have to do with the so-called "catcodes" of symbols used inside Ti*k*Z. In normal TeX operation, symbols like `!` or `"` are "normal" characters and the Ti*k*Z parser expects them to be. Some packages, most noticeably the `babel` package, aggressively change these character codes so that for instance a semicolon gets a little extra space in `french` mode or a quotation mark followed by a vertical bar breaks ligatures in `german` mode.

Unfortunately, Ti*k*Z expects the character codes of some symbols to be "normal". In some important cases it will tolerate changed character codes, but when the changes made by `babel` (or some other package) are too "aggressive", compilation of Ti*k*Z code will fail.

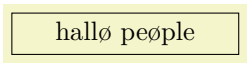
The `babel` library of Ti*k*Z is intended to help out in this situation. All this library does is to set the following two keys to `true`. You can, however, also set these keys directly and also switch them off or on individually and independently of this library.

`/tikz/handle active characters in code`=⟨*true or false*⟩ (no default, initially `false`)

When this key is set, at the beginning of every `\tikz` command and every `{tikzpicture}`, the character codes of all symbols used by Ti*k*Z are reset to their normal values. Furthermore, at the beginning of each node, the catcodes are restored to the values they had prior to the current picture.

The net effect of this is that, in most cases, symbols having a special character code can be used nicely both in Ti*k*Z code and also in node texts.

In the following, slightly silly, example we make the dot an active character and define it in some strange way. Now, in the later Ti*k*Z command, the dot in `3.0cm` may no longer be active and setting the `handle...` option achieves exactly this. However, as can be seen, the dot is once more active inside the node.



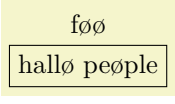

```
\catcode`\.=\active
\def.{\o}

\tikz [handle active characters in code]
  \node [draw, minimum width=3.0cm] {hall. pe.ple};
```

`/tikz/handle active characters in nodes`=⟨*true or false*⟩ (no default, initially `false`)

This key is needed for a special situation: As explained for the `handle ... code` key, that key switches off all special meaning of symbols and switches them back on again at the beginning of nodes. However, there is one situation when this is not possible: When some text has already been read by TeX, the catcodes can no longer change. Now, for normal nodes this is not a problem since their contents has not been read at the moment the catcodes are restored. In contrast for label nodes for edges, nodes produced by the `graph` and `quotes` libraries, and some others nodes, their text *has* already been read when the catcodes get adjusted.

The present key may help in such situations: It causes the text of all such "indirectly created" nodes to be surrounded by a call to the `\scantokens` command. This command attempts to reread an already read text, but allows catcodes to change. As users of this command will know, it is not a perfect substitute for directly reading the text by TeX, but it normally has the desired effect.




```
\catcode`\.=\active
\def.{\o}

\tikz [handle active characters in code,
       handle active characters in nodes]
  \node [draw, label=f..] {hall. pe.ple};
```

45 Background Library

TikZ Library backgrounds

```
\usetikzlibrary{backgrounds} % LaTeX and plain TeX
\usetikzlibrary[backgrounds] % ConTeXt
```

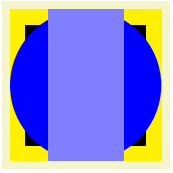
This library defines "backgrounds" for pictures. This does not refer to background pictures, but rather to frames drawn around and behind pictures. For example, this package allows you to just add the `framed` option to a picture to get a rectangular box around your picture or `gridded` to put a grid behind your picture.

The first use of this library is to make the following key available:

`/tikz/on background layer=`⟨*options*⟩ (default empty)

This key can (only) be used with a `{scope}` or `\scoped`. It will cause everything inside the scope to be typeset on a background layer.

The ⟨*options*⟩ will be executed *inside* background scope. This is useful since *other* options passed to the `{scope}` environment will be executed *before* the actual background material starts and, thus, will have no effect on it.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  % On main layer:
  \fill[blue] (0,0) circle [radius=1cm];

  \begin{scope}[on background layer={color=yellow}]
    \fill (-1,-1) rectangle (1,1);
  \end{scope}

  \begin{scope}[on background layer]
    \fill[black] (-.8,-.8) rectangle (.8,.8);
  \end{scope}

  % On main layer again:
  \fill[blue!50] (-.5,-1) rectangle (.5,1);
\end{tikzpicture}
```

A scope with this option set should not be "deeply nested" inside the picture since changes to the graphic state (like the color or the transformation matrix) "do not survive a layer switch", see also Section 113 for details. In particular, setting, say, the line width at the beginning of a picture will not have an effect on the background picture.

For this reason, it may be useful to setup the following style:

`/tikz/every on background layer` (style, no value)

This style is executed at the beginning of each background layer. If you have a global setup in `every picture`, you should consider putting that part of it that concerns the graphics state into this style.



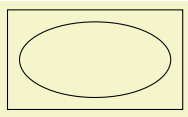
```
\usetikzlibrary {backgrounds}
\tikzset{
  every picture/.style={line width=1ex},
  every on background layer/.style={every picture}
}
\begin{tikzpicture}
  \draw [->] (0,0) -- (2,1);

  \scoped[on background layer]
    \draw[red] (0,1) -- (2,0);
\end{tikzpicture}
```

When this package is loaded, the following styles become available:

`/tikz/show background rectangle` (style, no value)

This style causes a rectangle to be drawn behind your graphic. This style option must be given to the `{tikzpicture}` environment or to the `\tikz` command.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[show background rectangle]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

The size of the background rectangle is determined as follows: We start with the bounding box of the picture. Then, a certain separator distance is added on the sides. This distance can be different for the x- and y-directions and can be set using the following options:

`/tikz/inner frame xsep=`⟨*dimension*⟩ (no default, initially `1ex`)

Sets the additional horizontal separator distance for the background rectangle.

`/tikz/inner frame ysep=`⟨*dimension*⟩ (no default, initially `1ex`)

Same for the vertical separator distance.

`/tikz/inner frame sep=`⟨*dimension*⟩ (no default)

Sets the horizontal and vertical separator distances simultaneously.

The following two styles make setting the inner separator a bit easier to remember:

`/tikz/tight background` (style, no value)

Sets the inner frame separator to 0pt. The background rectangle will have the size of the bounding box.

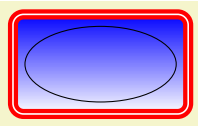
`/tikz/loose background` (style, no value)

Sets the inner frame separator to 2ex.

You can influence how the background rectangle is rendered by setting the following style:

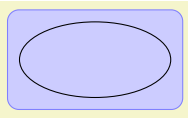
`/tikz/background rectangle` (style, initially `draw`)

This style dictates how the background rectangle is drawn or filled. The default setting causes the path of the background rectangle to be drawn in the usual way. Setting this style to, say, `fill=blue!20` causes a light blue background to be added to the picture. You can also use more fancy settings as shown in the following example:



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle/.style=
     {double,ultra thick,draw=red,top color=blue,rounded corners},
   show background rectangle]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

Naturally, no one in their right mind would use the above, but here is a nice background:



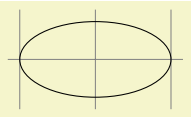
```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle/.style=
     {draw=blue!50,fill=blue!20,rounded corners=1ex},
   show background rectangle]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

`/tikz/framed` (style, no value)

This is a shorthand for `show background rectangle`.

`/tikz/show background grid` (style, no value)

This style behaves similarly to the `show background rectangle` style, but it will not use a rectangle path, but a grid. The lower left and upper right corner of the grid is computed in the same way as for the background rectangle:

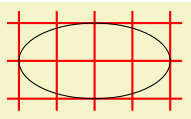


```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[show background grid]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

You can influence the background grid by setting the following style:

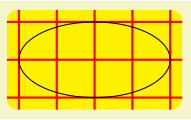
`/tikz/background grid` (style, initially `draw,help lines`)

This style dictates how the background grid path is drawn.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background grid/.style={thick,draw=red,step=.5cm},
   show background grid]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

This option can be combined with the `framed` option (use the `framed` option first):



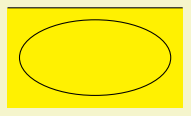
```
\usetikzlibrary {backgrounds}
\tikzset{background grid/.style={thick,draw=red,step=.5cm},
         background rectangle/.style={rounded corners,fill=yellow}}
\begin{tikzpicture}[framed,gridded]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

`/tikz/gridded` (style, no value)

This is a shorthand for `show background grid`.

`/tikz/show background top` (style, no value)

This style causes a single line to be drawn at the top of the background rectangle. Normally, the line coincides exactly with the top line of the background rectangle:

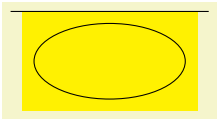


```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}[
    background rectangle/.style={fill=yellow},
    framed,show background top]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

The following option allows you to lengthen (or shorten) the line:

`/tikz/outer frame xsep=`⟨*dimension*⟩ (no default, initially `0pt`)

The ⟨*dimension*⟩ is added at the left and right side of the line.



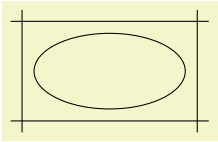
```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle/.style={fill=yellow},
   framed,
   show background top,
   outer frame xsep=1ex]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

`/tikz/outer frame ysep=`⟨*dimension*⟩ (no default, initially `0pt`)

This option does not apply to the top line, but to the left and right lines, see below.

`/tikz/outer frame sep=`⟨*dimension*⟩ (no default)

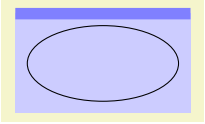
Sets both the x- and y-separation.



```
\usetikzlibrary {backgrounds}
\begin{tikzpicture}
  [background rectangle={fill=blue!20},
   outer frame sep=1ex,%
   show background top,%
   show background bottom,%
   show background left,%
   show background right]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

You can influence how the line is drawn grid by setting the following style:

`/tikz/background top` (style, initially `draw`)



```
\usetikzlibrary {backgrounds}
\tikzset{background rectangle/.style={fill=blue!20},
         background top/.style={draw=blue!50,line width=1ex}}
\begin{tikzpicture}[framed,show background top]
  \draw (0,0) ellipse [x radius=10mm, y radius=5mm];
\end{tikzpicture}
```

`/tikz/show background bottom` (style, no value)

Works like the style for the top line.

`/tikz/show background left` (style, no value)

Works similarly.

`/tikz/show background right` (style, no value)

Works similarly.

46 Calc Library

Ti*k*Z Library calc

```
\usetikzlibrary{calc} % LaTeX and plain TeX
\usetikzlibrary[calc] % ConTeXt
```

The library allows advanced Coordinate Calculations. It is documented in all detail in Section 13.5 on page 147.

47 Calendar Library

Ti*k*Z Library calendar

```
\usetikzlibrary{calendar} % LaTeX and plain TeX
\usetikzlibrary[calendar] % ConTeXt
```

The library defines the `\calendar` command, which can be used to typeset calendars. The command relies on the `\pgfcalendar` command from the `pgfcalendar` package, which is loaded automatically.

The `\calendar` command is quite configurable, allowing you to produce all kinds of different calendars.

47.1 Calendar Command

The core command for creating calendars in Ti*k*Z is the `\calendar` command. It is available only inside `{tikzpicture}` environments (similar to, say, the `\draw` command).

`\calendar`⟨*calendar specification*⟩`;`

The syntax for this command is similar to commands like `\node` or `\matrix`. However, it has its complete own parser and only those commands described in the following will be recognized, nothing else. Note, furthermore, that a ⟨*calendar specification*⟩ is not a path specification, indeed, no path is created for the calendar.

The specification syntax. The ⟨*calendar specification*⟩ must be a sequence of elements, each of which has one of the following structures:

- `[`⟨*options*⟩`]`
  You provide ⟨*options*⟩ in square brackets as in `[red,draw=none]`. These ⟨*options*⟩ can be any Ti*k*Z option and they apply to the whole calendar. You can provide this element multiple times, the effect accumulates.
- `(`⟨*name*⟩`)`
  This has the same effect as saying `[name=`⟨*name*⟩`]`. The effect of providing a ⟨*name*⟩ is explained later. Note already that *a calendar is not a node* and the ⟨*name*⟩ is *not the name of a node*.
- `at (`⟨*coordinate*⟩`)`
  This has the same effect as saying `[at=(`⟨*coordinate*⟩`)]`.
- `if (`⟨*date condition*⟩`)` ⟨*options or commands*⟩`else`⟨*else options or commands*⟩
  The effect of such an `if` is explained later.

At the beginning of every calendar, the following style is used:

`/tikz/every calendar` (style, initially empty)

This style is used with every calendar.

The date range. The overall effect of the `\calendar` command is to execute code for each day of a range of dates. This range of dates is set using the following option:

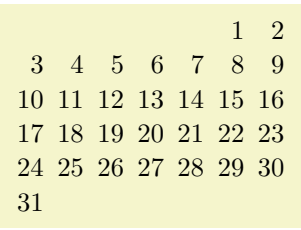
`/tikz/dates=`⟨*start date*⟩ `to` ⟨*end date*⟩ (no default)

This option specifies the date range. Both the start and end date are specified and described on page 1009. In short: You can provide ISO-format type dates like `2006-01-02`, you can replace the day of month by `last` to refer to the last day of a month (so `2006-02-last` is the same as `2006-02-28`), and you can add a plus sign followed by a number to specify an offset (so `2006-01-01+-1` is the same as `2005-12-31`).

It will be useful to fix two pieces of terminology for the following descriptions: The `\calendar` command iterates over the dates in the range. The *current date* refers to the current date the command is processing as it iterates over the dates. For each current date code is executed, which will be called the *current date code*. The current date code consists of different parts, to be detailed later.

The central part of the current date code is the execution of the code `\tikzdaycode`. By default, this code simply produces a node whose text is set to the day of month. This means that unless further action is taken, all days of a calendar will be put on top of each other! To avoid this, you must modify the current date code to shift days around appropriately. Predefined arrangements like `day list downward`

or `week list` do this for you, but you can define arrangements yourself. Since defining an arrangement is a bit tricky, it is explained only later on. For the time being, let us use a predefined arrangement to produce our first calendar:

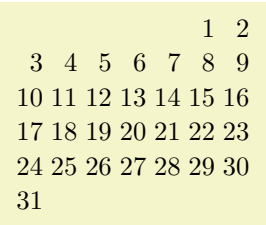


```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the spacing. In the above calendar, the spacing between the days is determined by numerous options. Most arrangements do not use all of these options, but only those that apply naturally.

`/tikz/day xshift=`⟨*dimension*⟩ (no default, initially `3.5ex`)

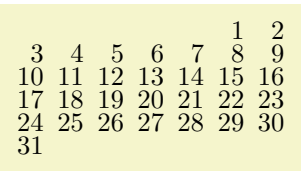
Specifies the horizontal shift between days. This is not the gap between days, but the shift between the anchors of their nodes.



```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,day xshift=3ex];
```

`/tikz/day yshift=`⟨*dimension*⟩ (no default, initially `3ex`)

Specifies the vertical shift between days. Again, this is the shift between the anchors of their nodes.



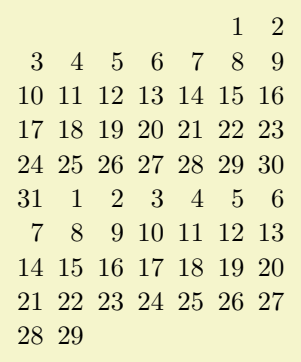
```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,day yshift=2ex];
```

`/tikz/month xshift=`⟨*dimension*⟩ (no default)

Specifies an additional horizontal shift between different months.

`/tikz/month yshift=`⟨*dimension*⟩ (no default)

Specifies an additional vertical shift between different months.



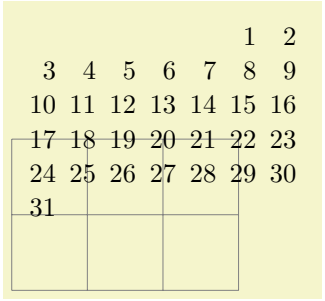
```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month yshift=0pt];
```



```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month yshift=1cm];
```

Changing the position of the calendar. The calendar is placed in such a way that, normally, the anchor of the first day label is at the origin. This can be changed by using the `at` option. When you say `at={(1,1)}`, this anchor of the first day will lie at coordinate $(1, 1)$.

In general, arrangements will not always place the anchor of the first day at the origin. Sometimes, additional spacing rules get in the way. There are different ways of addressing this problem: First, you can just ignore it. Since calendars are often placed in their own `{tikzpicture}` and since their size is computed automatically, the exact position of the origin often does not matter at all. Second, you can put the calendar inside a node as in `...node {\tikz \calendar...}`. This allows you to position the node in the normal ways using the node's anchors. Third, you can be very clever and use a single-cell matrix. The advantage is that a matrix allows you to provide any anchor of any node inside the matrix as an anchor for the whole matrix. For example, the following calendar is placed in such a way the center of 2000-01-20 lies on the position $(2, 2)$:



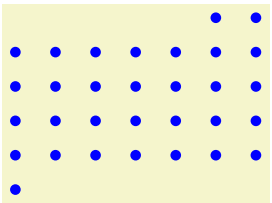
```
\usetikzlibrary {calendar}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \matrix [anchor=cal-2000-01-20.center] at (2,2)
  { \calendar(cal)[dates=2000-01-01 to 2000-01-31,week list]; \\};
\end{tikzpicture}
```

Unfortunately, the matrix-base positions, which is the cleanest way, isn't as portable as the other approaches (it currently does not work with the SVG backend for instance).

Changing the appearance of days. As mentioned before, each day in the above calendar is produced by an execution of the `\tikzdaycode`. Each time this code is executed, the coordinate system will have been set up appropriately to place the day of the month correctly. You can change both the code and its appearance using the following options.

`/tikz/day code=`⟨*code*⟩ (no default, initially see below)

This option allows you to change the code that is executed for each day. The default is to create a node with an appropriate name, but you can change this:



```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day code={\fill[blue] (0,0) circle [radius=2pt];}];
```

The default code is the following:

```
\node[name=\pgfcalendarsuggestedname,every day]{\tikzdaytext};
```

The first part causes the day nodes to be accessible via the following names: If ⟨*name*⟩ is the name given to the calendar via a `name=` option or via the specification element `(`⟨*name*⟩`)`, then `\pgfcalendarsuggestedname` will expand to ⟨*name*⟩`-`⟨*date*⟩, where ⟨*date*⟩ is the date of the day that is currently being processed in ISO format.

For example, if January 1, 2006 is being processed and the calendar has been named `mycal`, then the node containing the `1` for this date will be names `mycal-2006-01-01`. You can later reference this node.

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\usetikzlibrary {calendar}
\begin{tikzpicture}
  \calendar (mycal) [dates=2000-01-01 to 2000-01-31,week list];

  \draw[red] (mycal-2000-01-20) circle [radius=4pt];
\end{tikzpicture}
```

`/tikz/day text=`⟨*text*⟩ (no default)

This option changes the setting of the `\tikzdaytext`. By default, this macro simply yields the current day of month, but you can change it arbitrarily. Here is a silly example:

					x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x	x	x	x	x	x	x
x						

```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day text=x];
```

More useful examples are based on using the `\%` command. This command is redefined inside a `\pgfcalendar` to mean the same as `\pgfcalendarshorthand`. (The original meaning of `\%` is lost inside the calendar, you need to save if before the calendar if you really need it.)

The `\%` inserts the current day/month/year/day of week in a certain format into the text. The first letter following the `\%` selects the type (permissible values are `d`, `m`, `y`, `w`), the second letter specifies how the value should be displayed (`-` means numerically, `=` means numerically with leading space, `0` means numerically with leading zeros, `t` means textual, and `.` means textual, abbreviated). For example `\%d0` gives the day with a leading zero (for more details see the description of `\pgfcalendarshorthand` on page 1014).

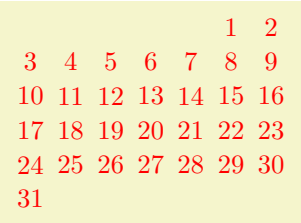
Let us redefine the `day text` so that it yields the day with a leading zero:

					01	02
03	04	05	06	07	08	09
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                day text=\%d0];
```

`/tikz/every day` (initially `anchor=base east`)

This style is executed by the default node code for each day. The `every day` style is useful for changing the way days look. For example, let us make all days red:



```
\usetikzlibrary {calendar}
\tikz[every day/.style=red]
  \calendar[dates=2000-01-01 to 2000-01-31,week list];
```

Changing the appearance of month and year labels. In addition to the days of a calendar, labels for the months and even years (for really long calendars) can be added. These labels are only added once per month or year and this is not done by default. Rather, special styles starting with `month label` place these labels and make them visible:



```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-02-last,week list,
                month label above centered];
```

The following options change the appearance of the month and year label:

`/tikz/month code=`⟨*code*⟩ (no default, initially see below)

This option allows you to specify what the macro `\tikzmonthcode` should expand to.

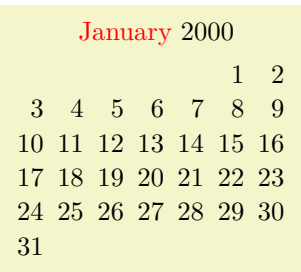
By default, the `\tikzmonthcode` it is set to

```
\node[every month]{\tikzmonthtext};
```

Note that this node is not named by default.

`/tikz/month text=`⟨*text*⟩ (no default)

This option allows you to change the macro `\tikzmonthtext`. By default, the month text is a long textual presentation of the current month being typeset.



```
\usetikzlibrary {calendar}
\tikz \calendar[dates=2000-01-01 to 2000-01-31,week list,
                month label above centered,
                month text=\textcolor{red}{\%mt} \%y];
```

`/tikz/every month` (style, initially empty)

This style can be used to change the appearance of month labels.

`/tikz/year code=`⟨*code*⟩ (no default)

Works like `month code`, only for years.

`/tikz/year text=`⟨*text*⟩ (no default)

Works like `month text`, only for years.

`/tikz/every year` (no value)

Works like `every month`, only for years.

Date ifs. Much of the power of the `\calendar` command comes from the use of conditionals. There are two equivalent way of specifying such a conditional. First, you can add the text `if (`⟨*conditions*⟩`)` ⟨*code or options*⟩ to your ⟨*calendar specification*⟩, possibly followed by `else`⟨*else code or options*⟩. You can have multiple such conditionals (but you cannot nest them in this simple manner). The second way is to use the following option:

`/tikz/if=(`⟨*conditions*⟩`)`⟨*code or options*⟩`else`⟨*else code or options*⟩ (no default)

This option has the same effect as giving a corresponding if in the ⟨*calendar specification*⟩. The option is mostly useful for use in the `every calendar` style, where you cannot provide if conditionals otherwise.

Now, regardless of how you specify a conditional, it has the following effect (individually and independently for each date in the calendar):

1. It is checked whether the current date is one of the possibilities listed in ⟨*conditions*⟩. An example of such a condition is `Sunday`. Thus, when you write `if (Saturday,Sunday) {foo}`, then `foo` will be executed for every day in the calendar that is a Saturday *or* a Sunday.

   The command `\ifdate` and, thereby, `\pgfcalendarifdate` are used to evaluate the ⟨*conditions*⟩, see page 1010 for a complete list of possible tests. The most useful tests are: Tests like `Monday` and so on, `workday` for the days Monday to Friday, `weekend` for Saturday and Sunday, `equals` for testing whether the current date equals a given date, `at least` and `at least` for comparing the current date with a given date.
2. If the date passes the check, the ⟨*code or options*⟩ is evaluated in a manner to be described in a moment; if the date fails, the ⟨*else code or options*⟩ is evaluated, if present.

   The ⟨*code or options*⟩ can either be some code. This is indicated by surrounding the code with curly braces. It can also be a list of Ti*k*Z options. This is indicated by surrounding the options with square brackets. For example in the date test `if (Sunday) {\draw...} else {\fill...}` there are two pieces of code involved. By comparison, `if (Sunday) [red] else [green]` involves two options.

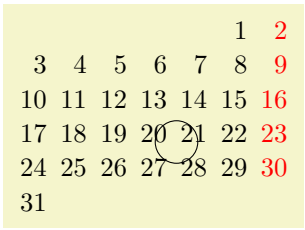
   If ⟨*code or options*⟩ is code, it is simply executed (for the current day). If it is a list of options, these options are passed to a scope surrounding the current date.

Let us now have a look at some examples. First, we use a conditional to make all Sundays red.

					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

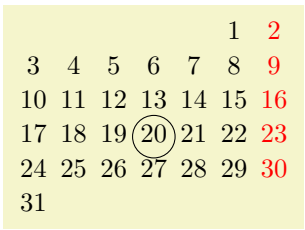
```
\usetikzlibrary {calendar}
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday) [red];
```

Next, let us do something on a specific date:



```
\usetikzlibrary {calendar}
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday)             [red]
    if (equals=2000-01-20) {\draw (0,0) circle [radius=8pt];};
```

You might wonder why the circle seems to be "off" the date. Actually, it is centered on the date, it is just that the date label uses the `base east` anchor, which shifts the label up and right. To overcome this problem we can change the anchor:



```
\usetikzlibrary {calendar}
\tikz [every day/.style={anchor=mid}]
  \calendar
    [dates=2000-01-01 to 2000-01-31,week list]
    if (Sunday)             [red]
    if (equals=2000-01-20) {\draw (0,0) circle [radius=8pt];};
```

However, the single day dates are now no longer aligned correctly. For this, we can change the day text to `\%d=`, which adds a space at the beginning of single day text.

In the following, more technical information is covered. Most readers may wish to skip it.

The current date code. As mentioned earlier, for each date in the calendar the current date code is executed. It is the job of this code to shift around date nodes, to render the date nodes, to draw the month labels and to do all other stuff that is necessary to draw a calendar.

The current date code consists of the following parts, in this order:

1. The before-scope code.
2. A scope is opened.
3. The at-begin-scope code.
4. All date-ifs from the ⟨*calendar specification*⟩ are executed.
5. The at-end-scope code.
6. The scope is closed.
7. The after-scope code.

All of the codes mentioned above can be changed using appropriate options, see below. In case you wonder why so many are needed, the reason is that the current date code as a whole is not surrounded by a scope or TEX group. This means that code executed in the before-scope code and in the after-scope code has an effect on all following days. For example, if the after-scope code modifies the transformation matrix by shifting everything downward, all following days will be shifted downward. If each day does this, you get a list of days, one below the other.

However, you do not always want code to have an effect on everything that follows. For instance, if a day has the date-if `if (Sunday) [red]`, we only want this Sunday to red, not all following days also. Similarly, sometimes it is easier to compute the position of a day relative to a fixed origin and we do not want any modifications of the transformation matrix to have an effect outside the scope.

By cleverly adjusting the different codes, all sorts of different day arrangements are possible.

`/tikz/execute before day scope=`⟨*code*⟩ (no default)

The ⟨*code*⟩ is executed before everything else for each date. Multiple calls of this option have an accumulative effect. Thus, if you use this option twice, the code from the first use is used first for each day, followed by the code given the second time.

`/tikz/execute at begin day scope=`⟨*code*⟩ (no default)

This code is execute before everything else inside the scope of the current date. Again, the effect is accumulative.

/tikz/execute at end day scope=⟨*code*⟩ (no default)

This code is executed just before the day scope is closed. The effect is also accumulative, however, in reverse order. This is useful to pair, say, `\scope` and `\endscope` commands in at-begin- and at-end-code.

/tikz/execute after day scope=⟨*code*⟩ (no default)

This is executed at the very end of the current date, outside the scope. The accumulation is also in reverse.

In the rest of the following subsections we have a look at how the different scope codes can be used to create different calendar arrangements.

47.1.1 Creating a Simple List of Days

We start with a list of the days of the calendar, one day below the other. For this, we simply shift the coordinate system downward at the end of the code for each day. This shift must be *outside* the day scope as we want day shifts to accumulate. Thus, we use the following code:

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-01 to 2000-01-08,
             execute after day scope=
               {\pgftransformyshift{-1em}}];
```

Clearly, we can use this approach to create day lists going up, down, right, left, or even diagonally.

47.1.2 Adding a Month Label

We now want to add a month label to the left of the beginning of each month. The idea is to do two things:

1. We add code that is executed only on the first of each month.
2. The code is executed before the actual day is rendered. This ensures that options applying to the days do not affect the month rendering.

We have two options where we should add the month code: Either we add it at the beginning of the day scope or before. Either will work fine, but it might be safer to put the code inside the scope to ensure that settings to not inadvertently "leak outside".

January

```
\usetikzlibrary {calendar}
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-08,
     execute after day scope={\pgftransformyshift{-1em}},
     execute at begin day scope=
       {\ifdate{day of month=1}{\tikzmonthcode}{}},
     every month/.append style={anchor=base east,xshift=-2em}];
```

In the above code we used the `\ifdate{`⟨*condition*⟩`}{`⟨*then code*⟩`}{`⟨*else code*⟩`}` command, which is described on page 1013 in detail and which has much the same effect as `if (`⟨*condition*⟩`){`⟨*then code*⟩`} else {`⟨*else code*⟩`}`, but works in normal code.

47.1.3 Creating a Week List Arrangement

Let us now address a more complicated arrangement: A week list. In this arrangement there is line for each week. The horizontal placement of the days is thus that all Mondays lie below each other, likewise for all Tuesdays, and so on.

In order to typeset this arrangement, we can use the following approach: The origin of the coordinate system rests at the anchor for the Monday of each week. That means that at the end of each week the origin

is moved downward one line. On all other days, the origin at the end of the day code is the same as at the beginning. To position each day correctly, we use code inside and at the beginning of the day scope to horizontally shift the day according to its day of week.

```
                1 2
 3 4 5 6 7 8 9
10111213141516
17181920
```

```
\usetikzlibrary {calendar}
\tikz
  \calendar
    [dates=2000-01-01 to 2000-01-20,
     % each day is shifted right according to the day of week
     execute at begin day scope=
       {\pgftransformxshift{\pgfcalendarcurrentweekday em}},
     % after each week, the origin is shifted downward:
     execute after day scope=
       {\ifdate{Sunday}{\pgftransformyshift{-1em}}{}}];
```

47.1.4 Creating a Month List Arrangement

For another example, let us create an arrangement that contains one line for each month. This is easy enough to do as for weeks, unless we add the following requirement: Again, we want all days in a column to have the same day of week. Since months start on different days of week, this means that each row has to have an individual offset.

One possible way is to use the following approach: After each month (or at the beginning of each month) we advance the vertical position of the offset by one line. For horizontal placement, inside the day scope we locally shift the day by its day of month. Furthermore, we must additionally shift the day to ensure that the first day of the month lies on the correct day of week column. For this, we remember this day of week the first time we see it.

```
         1 2 3 4 5 6 7 8 9101112131415161718192021222324252627282930 31
1 2 3 4 5 6 7 8 9101112131415161718192021222324252627282 9
```

```
\usetikzlibrary {calendar}
\newcount\mycount
\tikz
  \calendar
    [dates=2000-01-01 to 2000-02-last,
     execute before day scope=
     {
       \ifdate{day of month=1} {
         % Remember the weekday of first day of month
         \mycount=\pgfcalendarcurrentweekday
         % Shift downward
         \pgftransformyshift{-1em}
       }{}
     },
     execute at begin day scope=
     {
       % each day is shifted right according to the day of month
       \pgftransformxshift{\pgfcalendarcurrentday em}
       % and additionally according to the weekday of the first
       \pgftransformxshift{\the\mycount em}
     }];
```

47.2 Arrangements

An *arrangement* specifies how the days of calendar are arranged on the page. The `calendar` library defines a number of predefined arrangements.

We start with arrangements in which the days are listed in a long line.

`/tikz/day list downward` (style, no value)

This style causes the days of a month to be typeset one below the other. The shift between days is given by `day yshift`. Between month an additional shift of `month yshift` is added.

28
29
30
31

1
2
3

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em];
```

/tikz/day list upward (style, no value)

Works as above, only the list grows upward instead of downward.

3
2
1

31
30
29
28

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list upward,month yshift=1em];
```

/tikz/day list right (style, no value)

This style also works as before, but the list of days grows to the right. Instead of `day yshift` and `month yshift`, the values of `day xshift` and `month xshift` are used.

28 29 30 31 1 2 3

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list right,month xshift=1em];
```

/tikz/day list left (style, no value)

As above, but the list grows left.

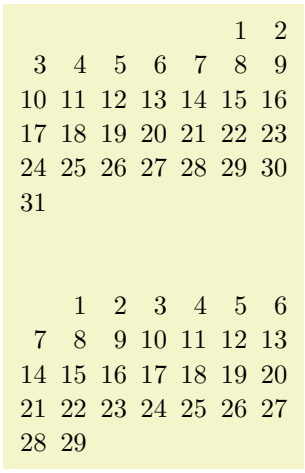
The next arrangement lists days by the week.

/tikz/week list (style, no value)

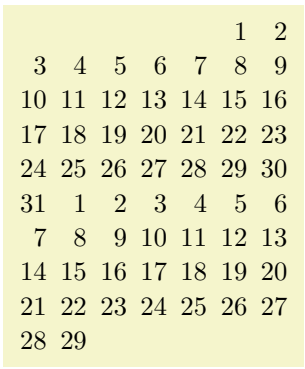
This style creates one row for each week in the range. The value of `day xshift` is used for the distance between days in each week row, the value of `day yshift` is used for the distance between rows. In both cases, "distance" refers to the distance between the anchors of the nodes of the days (or, more generally, the distance between the origins of the little pictures created for each day).

The days inside each week are shifted such that Monday is always at the first position (to change this, you need to copy and then modify the code appropriately). If the date range does not start on a Monday, the first line will not start in the first column, but rather in the column appropriate for the first date in the range.

At the beginning of each month (except for the first month in the range) an additional vertical space of `month yshift` is added. If this is set to `0pt` you get a continuous list of days.



```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list];
```



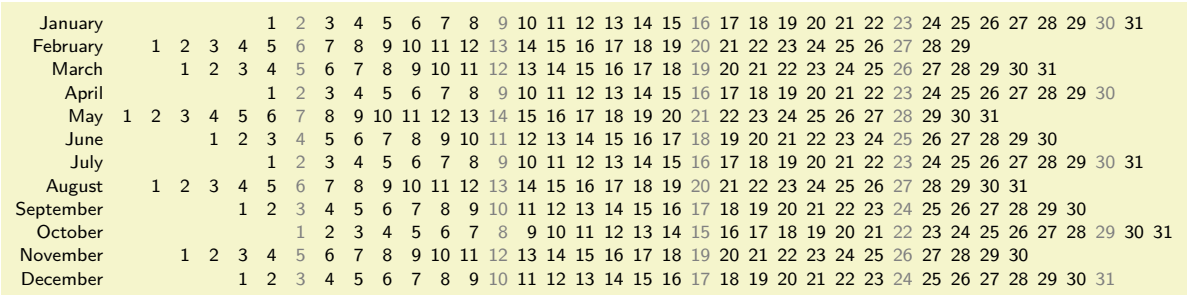
```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-01 to 2000-02-last,week list,
             month yshift=0pt];
```

The following arrangement gives a very compact view of a whole year.

`/tikz/month list` (style, no value)

In this arrangement there is a row for each month. As for the `week list`, the `day xshift` is used for the horizontal distance. For the vertical shift, `month yshift` is used.

In each row, all days of the month are listed alongside each other. However, it is once more ensured that days in each column lie on the same day of week. Thus, the very first column contains only Mondays. If a month does not start with a Monday, its days are shifted to the right such that the days lie on the correct columns.



```
\usetikzlibrary {calendar}
\sffamily\scriptsize
\tikz
  \calendar [dates=2000-01-01 to 2000-12-31,
             month list,month label left,month yshift=1.25em]
            if (Sunday) [black!50];
```

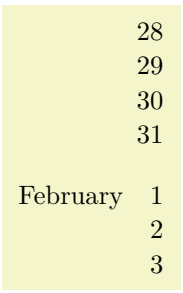
47.3 Month Labels

For many calendars you may wish to add a label to each month. We have already covered how month nodes are created and rendered in the description of the `\calendar` command: use `month text`, `every month`, and also `month code` (if necessary) to change the appearance of the month labels.

What we have not yet covered is where these labels are placed. By default, they are not placed at all as there is no good "default position" for them. Instead, you can use one of the following options to specify a position for the labels:

`/tikz/month label left` (style, no value)

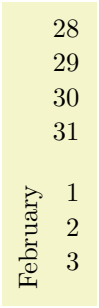
Places the month label to the left of the first day of the month. (For `week list` and `month list` where a month does not start on a Monday, the position is chosen "as if" the month had started on a Monday – which is usually exactly what you want.)




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em,
             month label left];
```

`/tikz/month label left vertical` (style, no value)

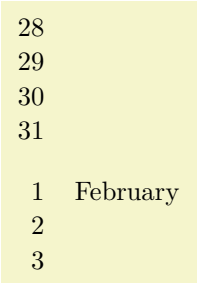
This style works like the above style, only the label is rotated counterclockwise by 90 degrees.




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em,
             month label left vertical];
```

`/tikz/month label right` (style, no value)

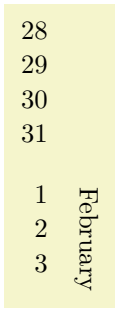
This style places the month label to the right of the row in which the first day of the month lies. This means that for a day list the label is to the right of the first day, for a week list it is to the right of the first week, and for a month list it is to the right of the whole month.




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em,
             month label right];
```

`/tikz/month label right vertical` (style, no value)

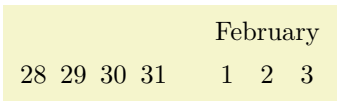
Works as above, only the label is rotated clockwise by 90 degrees.



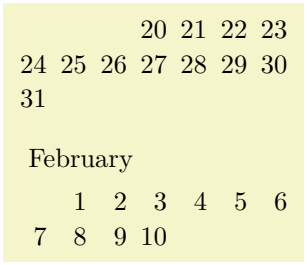

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list downward,month yshift=1em,
             month label right vertical];
```

/tikz/month label above left (style, no value)

This style places the month label above of the row of the first day, flushed left to the leftmost column. The amount by which the label is raised is fixed to `1.25em`; use the `yshift` option with the month node to modify this.



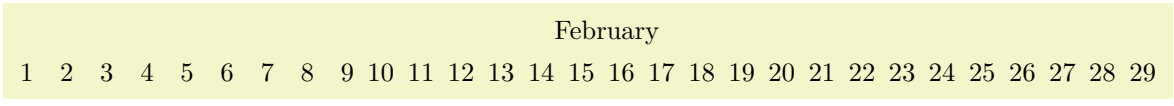

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-28 to 2000-02-03,
             day list right,month xshift=1em,
             month label above left];
```



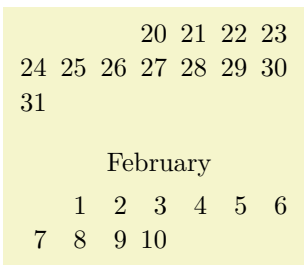

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above left];
```

/tikz/month label above centered (style, no value)

Works as above, only the label is centered above the row containing the first day.



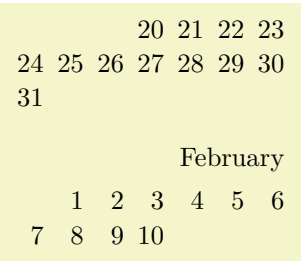

```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-02-01 to 2000-02-last,
             day list right,month label above centered];
```




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above centered];
```

/tikz/month label above right (style, no value)

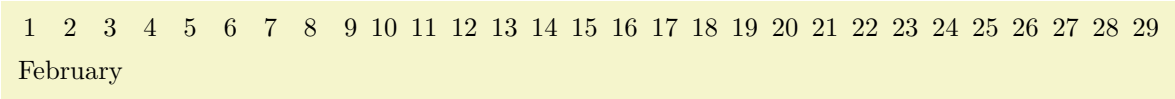
Works as above, but flushed right




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-01-20 to 2000-02-10,
             week list,month label above right];
```

/tikz/month label below left (style, no value)

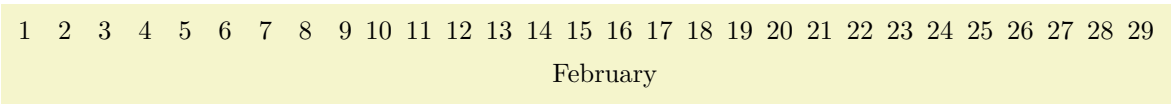
Works like `month label above left`, only the label is placed below the row. This placement is not really useful with the `week list` arrangement, but rather with the `day list right` or `month list` arrangement.




```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-02-01 to 2000-02-last,
             day list right,month label below left];
```

/tikz/month label below centered (style, no value)

Works like `month label above centered`, only below.

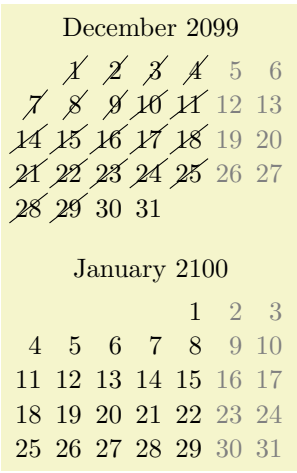



```
\usetikzlibrary {calendar}
\tikz
  \calendar [dates=2000-02-01 to 2000-02-last,
             day list right,month label below centered];
```

47.4 Examples

In the following, some example calendars are shown that come either from real applications or are just nice to look at.

Let us start with a year-2100-countdown, in which we cross out dates as we approach the big celebration. For this, we set the shape to `strike out` for these dates.



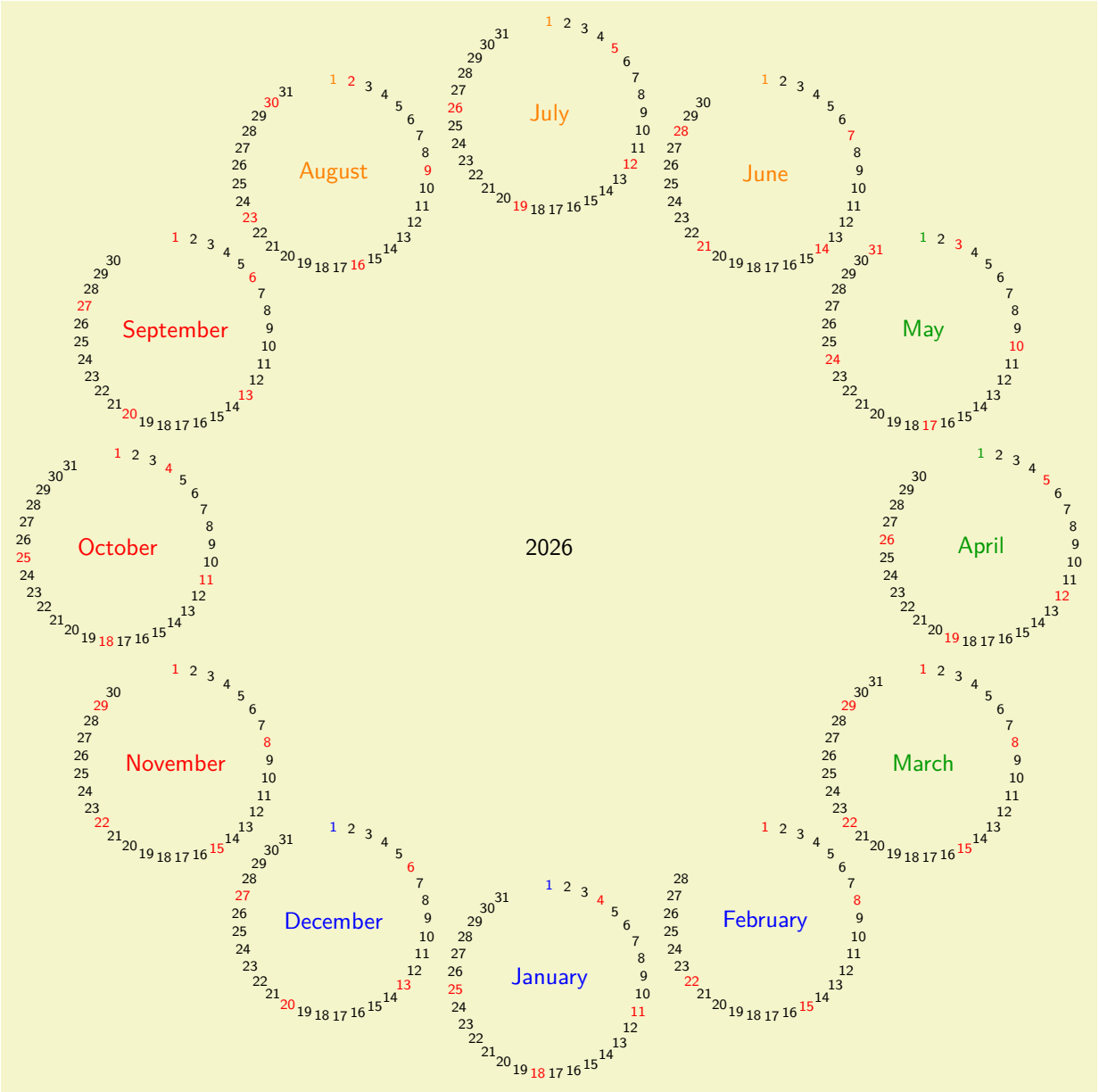

```
\usetikzlibrary {calendar,shapes.misc}
\begin{tikzpicture}
  \calendar
  [
    dates=2099-12-01 to 2100-01-last,
    week list,inner sep=2pt,month label above centered,
    month text=\%mt \%y0
  ]
  if (at most=2099-12-29) [nodes={strike out,draw}]
  if (weekend)            [black!50,nodes={draw=none}]
  ;
\end{tikzpicture}
```

The next calendar shows a deadline, which is 10 days in the future from the current date. The last three days before the deadline are in red, because we really should be done by then. All days on which we can no longer work on the project are crossed out.




```
\usetikzlibrary {calendar,shapes.misc}
\begin{tikzpicture}
  \calendar
  [
    dates=\year-\month-\day+-25 to \year-\month-\day+25,
    week list,inner sep=2pt,month label above centered,
    month text=\textit{\%mt \%y0}
  ]
  if (at least=\year-\month-\day) {}
    else [nodes={strike out,draw}]
  if (at most=\year-\month-\day+7)
    [green!50!black]
  if (between=\year-\month-\day+8 and \year-\month-\day+10)
    [red]
  if (Sunday)
    [gray,nodes={draw=none}]
  ;
\end{tikzpicture}
```

The following example is a futuristic calendar that is all about circles:


2026
January
February
March
April
May
June
July
August
September
October
November
December

```
\usetikzlibrary {calendar}
\sffamily

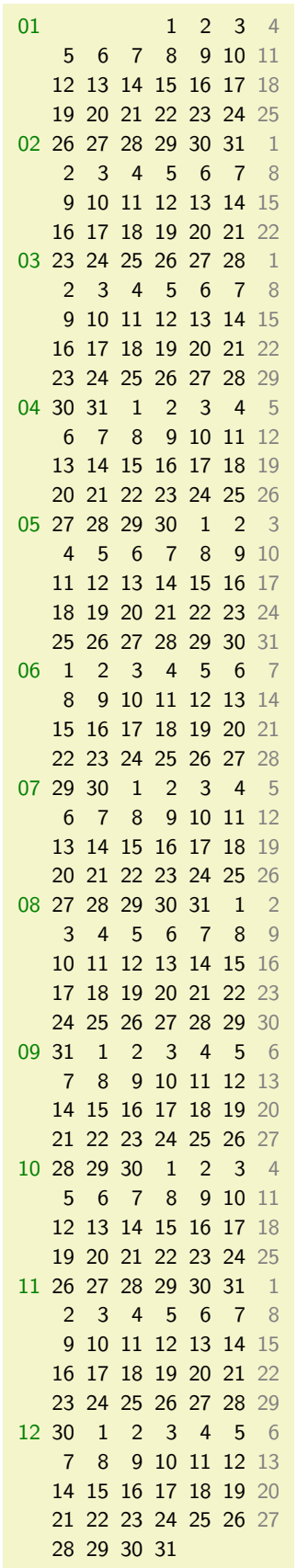
\colorlet{winter}{blue}
\colorlet{spring}{green!60!black}
\colorlet{summer}{orange}
\colorlet{fall}{red}

% A counter, since TikZ is not clever enough (yet) to handle
% arbitrary angle systems.
\newcount\mycount

\begin{tikzpicture}
  [transform shape,
   every day/.style={anchor=mid,font=\fontsize{6}{6}\selectfont}]
  \node{\normalsize\the\year};
  \foreach \month/\monthcolor in
    {1/winter,2/winter,3/spring,4/spring,5/spring,6/summer,
     7/summer,8/summer,9/fall,10/fall,11/fall,12/winter}
  {
    % Compute angle:
    \mycount=\month
    \advance\mycount by -1
    \multiply\mycount by 30
    \advance\mycount by -90

    % The actual calendar
    \calendar at (\the\mycount:6.4cm)
    [
      dates=\the\year-\month-01 to \the\year-\month-last,
    ]
    if (day of month=1) {\color{\monthcolor}\tikzmonthcode}
    if (Sunday) [red]
    if (all)
    {
      % Again, compute angle
      \mycount=1
      \advance\mycount by -\pgfcalendarcurrentday
      \multiply\mycount by 11
      \advance\mycount by 90
      \pgftransformshift{\pgfpointpolar{\mycount}{1.4cm}}
    };
  }
\end{tikzpicture}
```

Next, let's us have a whole year in a tight column:




```
\usetikzlibrary {calendar}
\begin{tikzpicture}
  \small\sffamily
  \colorlet{darkgreen}{green!50!black}
  \calendar[dates=\year-01-01 to \year-12-31,week list,
            month label left,month yshift=0pt,
            month text=\textcolor{darkgreen}{\%m0}]
           if (Sunday) [black!50];
\end{tikzpicture}
```

48 Chains

TikZ Library chains

```
\usetikzlibrary{chains} % LaTeX and plain TeX
\usetikzlibrary[chains] % ConTeXt
```

This library defines options for creating chains.

48.1 Overview

Chains are sequences of nodes that are – typically – arranged in a row or a column and that are – typically – connected by edges. More generally, they can be used to position nodes of a branching network in a systematic manner. For the positioning of nodes in rows and columns you can also use matrices, see Section 20, but chains can also be used to describe the connections between nodes that have already been connected using, say, matrices. Thus, it often makes sense to use matrices for the positioning of elements and chains to describe the connections.

48.2 Starting and Continuing a Chain

Typically, you construct one chain at a time, but it is permissible to construct multiple chains simultaneously. In this case, the chains must be named differently and you must specify for each node which chain it belongs to.

The first step toward creating a chain is to use the `start chain` option.

`/tikz/start chain`=⟨*chain name*⟩⟨*direction*⟩ (no default)

This key should, but need not, be given as an option to a scope enclosing all nodes of the chain. Typically, this will be a `scope` or the whole `tikzpicture`, but it might just be a path on which all nodes of the chain are found. If no ⟨*chain name*⟩ is given, the default value `chain` will be used instead.

The key starts a chain named ⟨*chain name*⟩ and makes it *active*, which means that it is currently being constructed. The `start chain` can be issued only once to activate a chain, inside a scope in which a chain is active you cannot use this option once more (for the same chain name). The chain stops being active at the end of the scope in which the `start chain` command was given.

Although chains are only locally active (that is, active inside the scope the `start chain` command was issued), the information concerning the chains is stored globally and it is possible to *continue* a chain after a scope has ended. For this, the `continue chain` option can be used, which allows you to reactivate an existing chain in another scope.

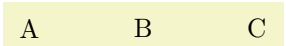
The ⟨*direction*⟩ is used to determine the placement rule for nodes on the chain. If it is omitted, the current value of the following key is used:

`/tikz/chain default direction`=⟨*direction*⟩ (no default, initially `going right`)

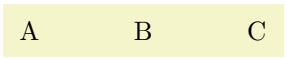
This ⟨*direction*⟩ is used in a `chain` option, if no other ⟨*direction*⟩ is specified.

The ⟨*direction*⟩ can have two different forms: `going` ⟨*options*⟩ or `placed` ⟨*options*⟩. The effect of these rules will be explained in the description of the `on chain` option. Right now, just remember that the ⟨*direction*⟩ you provide with the `chain` option applies to the whole chain.

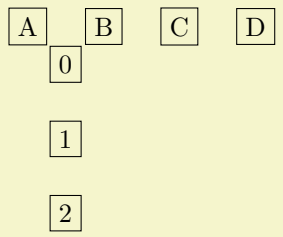
Other than this, this key has no further effect. In particular, to place nodes on the chain, you must use the `on chain` option, described next.




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain]
  % The chain is called just "chain"
  \node [on chain] {A};
  \node [on chain] {B};
  \node [on chain] {C};
\end{tikzpicture}
```




```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}
  % Same as above, using the scope shorthand
  { [start chain]
    \node [on chain] {A};
    \node [on chain] {B};
    \node [on chain] {C};
  }
\end{tikzpicture}
```




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=1 going right,
                    start chain=2 going below,
                    node distance=5mm,
                    every node/.style=draw]
  \node [on chain=1] {A};
  \node [on chain=1] {B};
  \node [on chain=1] {C};

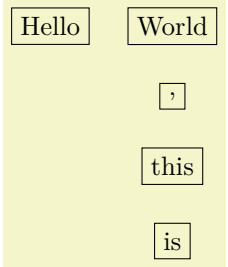
  \node [on chain=2] at (0.5,-.5) {0};
  \node [on chain=2] {1};
  \node [on chain=2] {2};

  \node [on chain=1] {D};
\end{tikzpicture}
```

/tikz/continue chain=⟨*chain name*⟩⟨*direction*⟩ (no default)

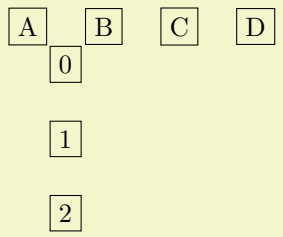
This option allows you to (re)activate an existing chain and to possibly change the default direction. If the `chain name` is missing, the name of the innermost activated chain is used. If no chain is activated, `chain` is used.

Let us have a look at the two different applications of this option. The first is to change the direction of a chain as it is being constructed. For this, just give this option somewhere inside the scope of the chain.




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=going right,node distance=5mm]
  \node [draw,on chain] {Hello};
  \node [draw,on chain] {World};
  \node [draw,continue chain=going below,on chain] {,};
  \node [draw,on chain] {this};
  \node [draw,on chain] {is};
\end{tikzpicture}
```

The second application is to reactivate a chain after it “has already been closed down”.




```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}[node distance=5mm,
                    every node/.style=draw]
  { [start chain=1]
    \node [on chain] {A};
    \node [on chain] {B};
    \node [on chain] {C};
  }

  { [start chain=2 going below]
    \node [on chain=2] at (0.5,-.5) {0};
    \node [on chain=2] {1};
    \node [on chain=2] {2};
  }

  { [continue chain=1]
    \node [on chain] {D};
  }
\end{tikzpicture}
```

48.3 Nodes on a Chain

`/tikz/on chain=`⟨*chain name*⟩⟨*direction*⟩ (no default)

This key should be given as an option to a node. When the option is used, the ⟨*chain name*⟩ must be the name of a chain that has been started using the `start chain` option. If ⟨*chain name*⟩ is the empty string, the current value of the innermost activated chain is used. If this option is used several times for a node, only the last invocation "wins". (To place a node on several chains, use the `\chainin` command repeatedly.)

The ⟨*direction*⟩ part is optional. If present, it sets the direction used for this node, otherwise the ⟨*direction*⟩ that was given to the original `start chain` option is used (or of the last `continue chain` option, which allows you to change this default).

The effects of this option are the following:

1. An internal counter (there is one local counter for each chain) is increased. This counter reflects the current number of the node in the chain, where the first node is node 1, the second is node 2, and so on.

   The value of this internal counter is globally stored in the macro `\tikzchaincount`.

2. If the node does not yet have a name, (having been given using the `name` option or the name-syntax), the name of the node is set to ⟨*chain name*⟩`-`⟨*value of the internal chain counter*⟩. For instance, if the chain is called `nums`, the first node would be named `nums-1`, the second `nums-2`, and so on. For the default chain name `chain`, the first node is named `chain-1`, the second `chain-2`, and so on.

3. Independently of whether the name has been provided automatically or via the `name` option, the name of the node is globally stored in the macro `\tikzchaincurrent`.

4. Except for the first node, the macro `\tikzchainprevious` is now globally set to the name of the node of the previous node on the chain. For the first node of the chain, this macro is globally set to the empty string.

5. Except possibly for the first node of the chain, the placement rule is now executed. The placement rule is just a Ti*k*Z option that is applied automatically to each node on the chain. Depending on the form of the ⟨*direction*⟩ parameter (either the locally given one or the one given to the `start chain` option), different things happen.

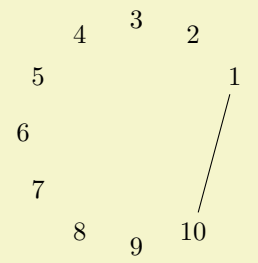
   First, it makes a difference whether the ⟨*direction*⟩ starts with `going` or with `placed`. The difference is that in the first case, the placement rule is not applied to the first node of the chain, while in the second case the placement rule is applied also to this first node. The idea is that a `going`-direction indicates that we are "going somewhere relative to the previous node" whereas a `placed` indicates that we are "placing nodes according to their number".

   Independently of which form is used, the ⟨*text*⟩ inside ⟨*direction*⟩ that follows `going` or `placed` (separated by a compulsory space) can have two different effects:

   (a) If it contains an equal sign, then this ⟨*text*⟩ is used as the placement rule, that is, it is simply executed.

   (b) If it does not contain an equal sign, then ⟨*text*⟩`=of \tikzchainprevious` is used as the placement rule.

   Note that in the first case, inside the ⟨*text*⟩ you have access to `\tikzchainprevious` and `\tikzchaincount` for doing your positioning calculations.




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=circle placed {at=(\tikzchaincount*30:1.5)}]
  \foreach \i in {1,...,10}
    \node [on chain] {\i};

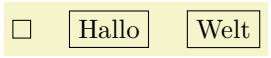
  \draw (circle-1) -- (circle-10);
\end{tikzpicture}
```

6. The following style is executed:

`/tikz/every on chain` (style, no value)

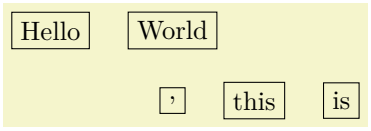
This key is executed for every node on a chain, including the first one.

Recall that the standard placement rule has a form like `right=of (\tikzchainprevious)`. This means that each new node is placed to the right of the previous one, spaced by the current value of `node distance`.



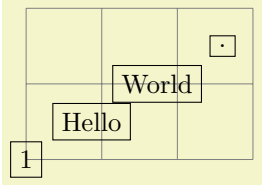

```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain,node distance=5mm]
  \node [draw,on chain] {};
  \node [draw,on chain] {Hallo};
  \node [draw,on chain] {Welt};
\end{tikzpicture}
```

The optional ⟨*direction*⟩ allows us to temporarily change the direction in the middle of a chain:




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain,node distance=5mm]
  \node [draw,on chain] {Hello};
  \node [draw,on chain] {World};
  \node [draw,on chain=going below] {,};
  \node [draw,on chain] {this};
  \node [draw,on chain] {is};
\end{tikzpicture}
```

You can also use more complicated computations in the ⟨*direction*⟩:




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain=going {at=(\tikzchainprevious),shift=(30:1)}]
  \draw [help lines] (0,0) grid (3,2);
  \node [draw,on chain] {1};
  \node [draw,on chain] {Hello};
  \node [draw,on chain] {World};
  \node [draw,on chain] {.};
\end{tikzpicture}
```

For each chain, two special "pseudo nodes" are created.

Predefined node ⟨*chain name*⟩`-begin`

This node is the same as the first node on the chain. It is only defined after a first node has been defined.

Predefined node ⟨*chain name*⟩`-end`

This node is the same as the (currently) last node on the chain. As the chain is extended, this node changes.

The `on chain` option can also be used, in conjunction with `late options`, to add an already existing node to a chain. The following command, which is only defined inside scopes where a `start chain` option is present, simplifies this process.

`\chainin`(⟨*existing name*⟩) [⟨*options*⟩]

This command makes it easy to add a node to chain that has already been constructed. This node may even be part of a another chain.

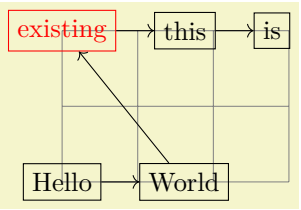
When you say `\chainin (some node);`, the node `some node` must already exist. It will then be made part of the current chain. This does not mean that the node can be changed (it is already constructed, after all), but the `join` option can be used to join `some node` to the previous last node on the chain and subsequent nodes will be placed relative to `some node`.

It is permissible to give the `on chain` option inside the ⟨*options*⟩ in order to specify on which chain the node should be put.

This command is just a shortcut for

```
\path (⟨existing name⟩) [late options={on chain,every chain in,⟨options⟩}]
```

In particular, it is possible to continue to path after a `\chainin` command, though that does not seem very useful.

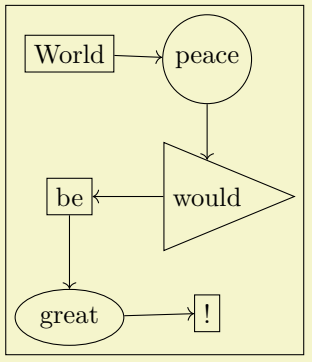



```
\usetikzlibrary {chains}
\begin{tikzpicture}[node distance=5mm,
                    every node/.style=draw,every join/.style=->]
  \draw [help lines] (0,0) grid (3,2);

  \node[red] (existing) at (0,2) {existing};

  \begin{scope}[start chain]
    \node [draw,on chain,join] {Hello};
    \node [draw,on chain,join] {World};
    \chainin (existing) [join];
    \node [draw,on chain,join] {this};
    \node [draw,on chain,join] {is};
  \end{scope}
\end{tikzpicture}
```

Here is an example where nodes are positioned using a matrix and then connected using a chain




```
\usetikzlibrary {chains,matrix,scopes,shapes.geometric}
\begin{tikzpicture}[every node/.style=draw]
  \matrix [matrix of nodes,column sep=5mm,row sep=5mm]
  {
    |(a)|           World & |(b) [circle]|             peace \\
    |(c)|           be    & |(d) [isosceles triangle]| would \\
    |(e) [ellipse]| great & |(f)|                      ! \\
  };

  % (the `scopes' library needs to be loaded to make the following work)
  { [start chain,every on chain/.style={join=by ->}]
    \chainin (a);
    \chainin (b);
    \chainin (d);
    \chainin (c);
    \chainin (e);
    \chainin (f);
  }
\end{tikzpicture}
```

48.4 Joining Nodes on a Chain

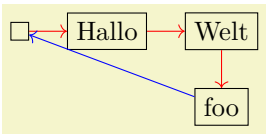
`/tikz/join=with` ⟨*with*⟩ `by` ⟨*options*⟩ (no default)

When this key is given to any node on a chain (except possibly for the first node), an `edge` command is added after the node. The `with` part specifies which node should be used for the start point of the edge; if the `with` part is omitted, the `\tikzchainprevious` is used. This `edge` command gets the ⟨*options*⟩ as parameter and the current node as its target. If there is no previous node and no `with` is given, no `edge` command gets executed.

`/tikz/every join` (style, no value)

This style is executed each time this command is used.

Note that it makes sense to call this option several times for a node, in order to connect it to several nodes. This is especially useful for joining in branches, see the next section.




```
\usetikzlibrary {chains}
\begin{tikzpicture}[start chain,node distance=5mm,
                    every join/.style={->,red}]
  \node [draw,on chain,join] {};
  \node [draw,on chain,join] {Hallo};
  \node [draw,on chain,join] {Welt};
  \node [draw,on chain=going below,
         join,join=with chain-1 by {blue,<-}] {foo};
\end{tikzpicture}
```

48.5 Branches

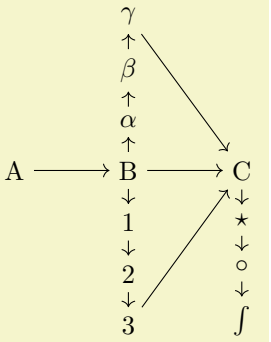
A *branch* is a chain that (typically only temporarily) extends an existing chain. The idea is the following: Suppose we are constructing a chain and at some node `x` there is a fork. In this case, one (or even more) branches starts at this fork. For each branch a chain is created, but the first node on this chain should be `x`. For this, it is useful to use `\chainin` on the node `x` to make it part of the different branch chains and to name the branch chains in some way that reflects the name of the main chain.

The `start branch` option provides a shorthand for doing exactly what was just described.

`/tikz/start branch=`⟨*branch name*⟩⟨*direction*⟩ (no default)

This key is used in the same manner as the `start chain` command, however, the effect is slightly different:

- This option may only be used if some chain is already active and there is a (last) node on this chain. Let us call this node the ⟨*fork node*⟩.
- The chain is not just called ⟨*branch name*⟩, but ⟨*current chain*⟩`/`⟨*branch name*⟩. For instance, if the ⟨*fork node*⟩ is part of the chain called `trunk` and the ⟨*branch name*⟩ is set to `left`, the complete chain name of the branch is `trunk/left`. The ⟨*branch name*⟩ must be given, there is no default value.
- The ⟨*fork node*⟩ is automatically "chained into" the branch chain as its first node. Thus, for the first node on the branch that you provide, the `join` option will cause it to be connected to the fork node.



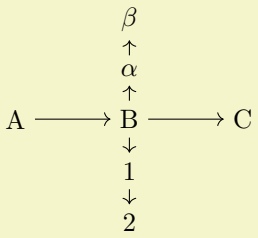

```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}[every on chain/.style=join,every join/.style=->,
                    node distance=2mm and 1cm]
  { [start chain=trunk]
    \node [on chain] {A};
    \node [on chain] {B};

    { [start branch=numbers going below]
      \node [on chain] {1};
      \node [on chain] {2};
      \node [on chain] {3};
    }
    { [start branch=greek going above]
      \node [on chain] {$\alpha$};
      \node [on chain] {$\beta$};
      \node [on chain] {$\gamma$};
    }

    \node [on chain,join=with trunk/numbers-end,join=with trunk/greek-end] {C};
    { [start branch=symbols going below]
      \node [on chain] {$\star$};
      \node [on chain] {$\circ$};
      \node [on chain] {$\int$};
    }
  }
\end{tikzpicture}
```

`/tikz/continue branch=`⟨*branch name*⟩⟨*direction*⟩ (no default)

This option works like the `continue chain` option, only ⟨*current chain*⟩`/`⟨*branch name*⟩ is used as the chain name, rather than just ⟨*branch name*⟩.




```
\usetikzlibrary {chains,scopes}
\begin{tikzpicture}[every on chain/.style=join,every join/.style=->,
                    node distance=2mm and 1cm]
  { [start chain=trunk]
    \node [on chain] {A};
    \node [on chain] {B};
    { [start branch=numbers going below] } % just a declaration,
    { [start branch=greek   going above] } % we will come back later
    \node [on chain] {C};

    % Now come the branches...
    { [continue branch=numbers]
      \node [on chain] {1};
      \node [on chain] {2};
    }
    { [continue branch=greek]
      \node [on chain] {$\alpha$};
      \node [on chain] {$\beta$};
    }
  }
\end{tikzpicture}
```

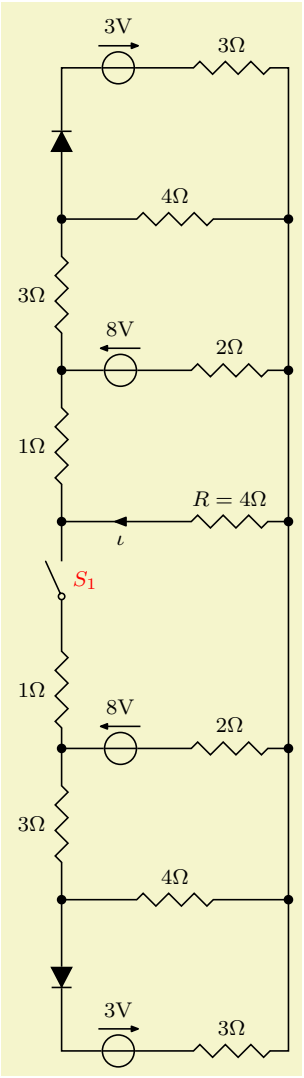
49 Circuit Libraries

Written and documented by Till Tantau, and Mark Wibrow. Inspired by the work of Massimo Redaelli.

49.1 Introduction

The circuit libraries can be used to draw different kinds of electrical or logical circuits. There is not a single library for this, but a whole hierarchy of libraries that work in concert. The main design goal was to create a balance between ease-of-use and ease-of-extending, while creating high-quality graphical representations of circuits.

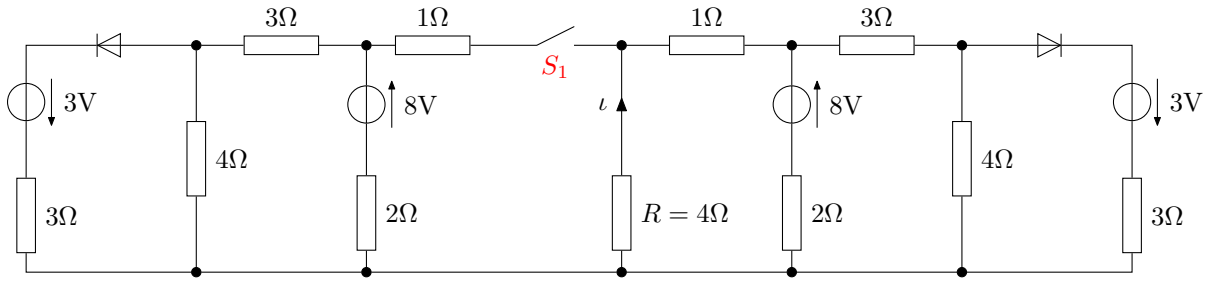
49.1.1 A First Example




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,x=3cm,y=2cm,semithick,
                    every info/.style={font=\footnotesize},
                    small circuit symbols,
                    set resistor graphic=var resistor IEC graphic,
                    set diode graphic=var diode IEC graphic,
                    set make contact graphic= var make contact IEC graphic]
  % Let us start with some contacts:
  \foreach \contact/\y in {1/1,2/2,3/3.5,4/4.5,5/5.5}
  {
    \node [contact] (left contact \contact) at (0,\y) {};
    \node [contact] (right contact \contact) at (1,\y) {};
  }
  \draw (right contact 1) -- (right contact 2) -- (right contact 3)
     -- (right contact 4) -- (right contact 5);

  \draw (left contact 1) to [diode] ++(down:1)
                         to [voltage source={near start,
                                             direction info={volt=3}},
                             resistor={near end,ohm=3}] ++(right:1)
                         to (right contact 1);
  \draw (left contact 1) to [resistor={ohm=4}] (right contact 1);
  \draw (left contact 1) to [resistor={ohm=3}] (left contact 2);
  \draw (left contact 2) to [voltage source={near start,
                                             direction info={<-,volt=8}},
                             resistor={ohm=2,near end}] (right contact 2);
  \draw (left contact 2) to [resistor={near start,ohm=1},
                             make contact={near end,info'={[red]$S_1$}}]
                         (left contact 3);
  \draw (left contact 3) to [current direction'={near start,info=$\iota$},
                             resistor={near end,info={$R=4\Omega$}}]
                         (right contact 3);
  \draw (left contact 4) to [voltage source={near start,
                                             direction info={<-,volt=8}},
                             resistor={ohm=2,near end}] (right contact 4);
  \draw (left contact 3) to [resistor={ohm=1}] (left contact 4);
  \draw (left contact 4) to [resistor={ohm=3}] (left contact 5);
  \draw (left contact 5) to [resistor={ohm=4}] (right contact 5);
  \draw (left contact 5) to [diode] ++(up:1)
                         to [voltage source={near start,
                                             direction info={volt=3}},
                             resistor={near end,ohm=3}] ++(right:1)
                         to (right contact 5);
\end{tikzpicture}
```

An important feature of the `circuits` library is that the appearance of a circuit can be configured in general ways and that the labels are placed automatically by default. Here is the graphic once more, generated from *exactly the same source code*, with only the options of the `{tikzpicture}` environment replaced by `[rotate=-90,circuit ee IEC,x=3.25cm,y=2.25cm]`:



49.1.2 Symbols

A circuit typically consists of numerous electronic elements like logical gates or resistors or diodes that are connected by wires. In PGF/Ti*k*Z, we use nodes for the electronic elements and normal lines for the wires. Ti*k*Z offers a large number of different ways of positioning and connecting nodes in general, all of which can be used here. Additionally, the `circuits` library defines an additional useful `to`-path that is particularly useful for elements like a resistor on a line.

There are many different names that are used to refer to electrical "elements", so a bit of terminology standardization is useful: We will call such elements *symbols*. A *symbol shape* is a PGF shape declared using the `\pgfdeclareshape` command. A *symbol node* is a node whose shape is a symbol shape.

49.1.3 Symbol Graphics

Symbols can be created by `\node[shape=some symbol shape]`. However, in order to represent some symbols correctly, just using standard PGF shapes is not sufficient. For instance, most symbols have a visually appealing "default size", but the size of a symbol shape depends only on the current values of parameters like `minimum height` or `inner xsep`.

For these reasons, the circuit libraries introduce the concept of a *symbol graphic*. This is a style that causes a `\node` to not only have the correct shape, but also the correct size and the correct path usage. More generally, this style may set up things in any way so that the "symbol looks correct". When you write, for instance, `\node[diode]`, then the style called `diode graphic` is used, which in turn is set to something like `shape=diode IEC,draw,minimum height=...`.

Here is an overview of the different kinds of circuit libraries:

- The Ti*k*Z-library `circuits` defines general keys for creating circuits. Mostly, these keys are useful for defining more specialized libraries.

  You normally do not use this library directly since it does not define any symbol graphics.

- The Ti*k*Z-library `circuits.logic` defines keys for creating logical gates like and-gates or xor-gates. However, this library also does not actually define any symbol graphics; this is done by two sublibraries:

  - The library `circuits.logic.US` defines symbol graphics that cause the logical gates to be rendered in the "US-style". It includes all of the above libraries and you can use this library directly.
  - The library `circuits.logic.IEC` also defines symbol graphics for logical gates, but it uses rectangular gates rather that the round US-gates. This library can coexist peacefully with the above library, you can change which symbol graphics are used "on the fly".

- The Ti*k*Z-library `circuits.ee` defines keys for symbols from electrical engineering like resistors or capacitors. Again, sublibraries define the actual symbol graphics.

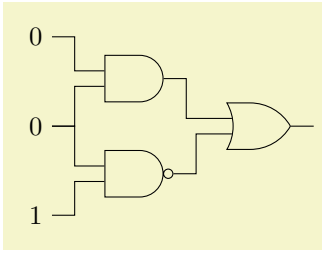
  - The library `circuits.ee.IEC` defines symbol shapes that follow the IEC norm.

- The PGF-libraries `shapes.gates.*` define (circuit) symbol shapes. However, you normally do not use these shapes directly, rather you use a style that uses an appropriate symbol graphic, which in turn uses one of these shapes.

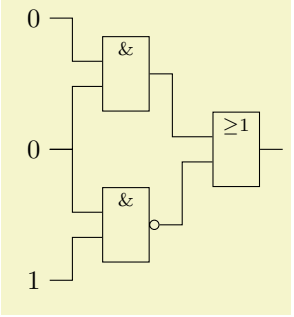
Let us have a look at a simple example. Suppose we wish to create a logical circuit. Then we first have to decide which symbol graphics we would like to use. Suppose we wish to use the US-style, then we would include the library `circuits.logic.US`. If you wish to use IEC-style symbols, use `circuits.logic.IEC`. If you cannot decide, include both:

```
\usetikzlibrary{circuits.logic.US,circuits.logic.IEC}
```

To create a picture that contains a US-style circuit you can now use the option `circuit logic US`. This will set up keys like `and gate` to create use an appropriate symbol graphic for rendering an `and gate`. Using the `circuit logic IEC` instead will set up `and gate` to use another symbol graphic.




```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture}[circuit logic US]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                                 & \\
                    & \node [and gate] (a1) {};       & \\
    \node (i1) {0}; &                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {};      & \\
    \node (i2) {1}; &                                 & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

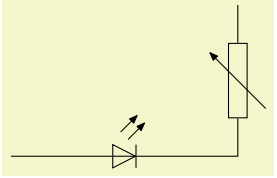



```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                                 & \\
                    & \node [and gate] (a1) {};       & \\
    \node (i1) {0}; &                                 & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {};      & \\
    \node (i2) {1}; &                                 & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

49.1.4 Annotations

An *annotation* is a little extra drawing that can be added to a symbol. For instance, when you add two little parallel arrows pointing away from some electrical element, this usually means that the element is light emitting.

Instead of having one symbol for "diode" and another for "light emitting diode", there is just one `diode` symbol, but you can add the `light emitting` annotation to it. This is done by passing the annotation as a parameter to the symbol as in the following example:



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \draw (0,0) to [diode={light emitting}] (3,0)
              to [resistor={adjustable}]  (3,2);
```

49.2 The Base Circuit Library

TikZ Library `circuits`

```
\usetikzlibrary{circuits} % LaTeX and plain TeX
\usetikzlibrary[circuits] % ConTeXt
```

This library is a base library that is included by other circuit libraries. You do not include it directly, but you will typically use some of the general keys, described below.

/tikz/circuits (no value)

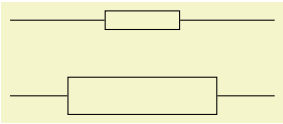
This key should be passed as an option to a picture or a scope that contains a circuit. It will do some internal setups. This key is normally called by more specialized keys like `circuit ee IEC`.

49.2.1 Symbol Size

/tikz/circuit symbol unit=⟨*dimension*⟩ (no default, initially `7pt`)

This dimension is a "unit" for the size of symbols. The libraries generally define the sizes of symbols relative to this dimension. For instance, the longer side of an inductor is, by default, in the IEC library equal to five times this ⟨*dimension*⟩. When you change this ⟨*dimension*⟩, the size of all symbols will automatically change accordingly.

Note, that it is still possible to overwrite the size of any particular symbol. These settings apply only to the default sizes.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (3.5,1);
  \draw[circuit symbol unit=14pt]
        (0,0) to [resistor] (3.5,0);
\end{tikzpicture}
```

/tikz/huge circuit symbols (style, no value)

This style sets the default circuit symbol unit to `10pt`.

/tikz/large circuit symbols (style, no value)

This style sets the default circuit symbol unit to `8pt`.

/tikz/medium circuit symbols (style, no value)

This style sets the default circuit symbol unit to `7pt`.

/tikz/small circuit symbols (style, no value)

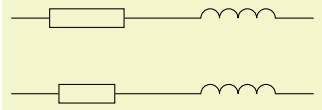
This style sets the default circuit symbol unit to `6pt`.

/tikz/tiny circuit symbols (style, no value)

This style sets the default circuit symbol unit to `5pt`.

/tikz/circuit symbol size=`width` ⟨*width*⟩ `height` ⟨*height*⟩ (no default)

This key sets `minimum height` to ⟨*height*⟩ times the current value of the circuit symbol unit and the `minimum width` to ⟨*width*⟩ times this value. Thus, this option can be used with a node command to set the size of the node as a multiple of the circuit symbol unit.



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
  \draw (0,1) to [resistor] (2,1) to[inductor] (4,1);

  \begin{scope}
    [every resistor/.style={circuit symbol size=width 3 height 1}]
    \draw (0,0) to [resistor] (2,0) to[inductor] (4,0);
  \end{scope}
\end{tikzpicture}
```

49.2.2 Declaring New Symbols

/tikz/circuit declare symbol=⟨*name*⟩ (no default)

This key is used to declare a symbol. It does not cause this symbol to be shown nor does it set a graphic to be used for the symbol, it simply "prepares" several keys that can later be used to draw a symbol and to configure it.

In detail, the first key that is defined is just called ⟨*name*⟩. This key should be given as an option to a `node` or on a `to` path, as explained below. The key will take options, which can be used to influence the way the symbol graphic is rendered.

Let us have a look at an example. Suppose we want to define a symbol called `foo`, which just looks like a simple rectangle. We could then say

```
\tikzset{circuit declare symbol=foo}
```

The symbol could now be used like this:

```
\node [foo]       at (1,1) {};
\node [foo={red}] at (2,1) {};
```

However, in the above example we would not actually see anything since we have not yet set up the graphic to be used by `foo`. For this, we must use a key called `set foo graphic` or, generally, `set` ⟨*name*⟩ `graphic`. This key gets graphic options as parameter that will be set when a symbol `foo` should be shown:



```
\usetikzlibrary {circuits}
\begin{tikzpicture}
  [circuit declare symbol=foo,
   set foo graphic={draw,shape=rectangle,minimum size=5mm}]

  \node [foo]       at (1,1) {};
  \node [foo={red}] at (2,1) {};
\end{tikzpicture}
```

In detail, when you use the key ⟨*name*⟩=⟨*options*⟩ with a node, the following happens:

1. The `inner sep` is set to `0.5pt`.
2. The following style is executed:

   `/tikz/every circuit symbol` (style, no value)

   Use this style to set up things in general.
3. The graphic options that have been set using `set` ⟨*name*⟩ `graphic` are set.
4. The style `every` ⟨*name*⟩ is executed. You can use it to configure the symbol further.
5. The ⟨*options*⟩ are executed.

The key ⟨*name*⟩ will have a different effect when it is used on a `to` path command inside a `circuit` environment (the `circuit` environment sets up `to` paths in such a way that the use of a key declared using `circuit declare symbol` is automatically detected). When ⟨*name*⟩ is used on a `to` path, the above actions also happen (setting the inner separation, using the symbol graphic, and so on), but they are passed to the key `circuit handle symbol`, which is explained next.

`/tikz/circuit handle symbol`=⟨*options*⟩ (no default)

This key is mostly used internally. Its purpose is to render a symbol. The effect of this key differs, depending on whether it is used as the optional argument of a `to` path command or elsewhere.

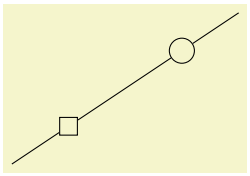
If the key is not used as an argument of a `to` path command, the ⟨*options*⟩ are simply executed.

The more interesting case happens when the key is given on a `to` path command. In this case, several things happen:

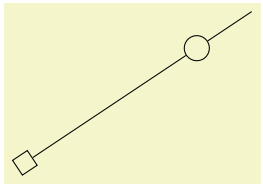
1. The `to` path is locally changed and set to an internal path (which you should not try to change) that consists mostly of a single straight line.
2. The ⟨*options*⟩ are tentatively executed with filtering switched on. Everything is filtered out, except for the key `pos` and also the styles `at start`, `very near start`, `near start`, `midway`, `near end`, `very near end`, and `at end`. If none of them is found, `midway` is used.
3. The filtered option is used to determine a position for the symbol on the path. At the given position (with `pos=0` representing the start and `pos=1` representing the end), a node will be added to the path (in a manner to be described presently).
4. This node gets ⟨*options*⟩ as its option list.
5. The node is added by virtue of a special `markings` decoration. This means that a `mark` command is executed that causes the node to be placed as a mark on the path.

6. The marking decoration will automatically subdivide the path and cause a line to be drawn from the start of the path to the node's border (at the position that lies on a line from the node's center to the start of the path) and then from the node's border (at a position on the other side of the node) to the end of the path.
7. The marking decoration will also take care of the case that multiple marks are present on a path, in this case the lines from and to the borders of the nodes are only between consecutive nodes.
8. The marking decoration will also rotate the coordinate system in such a way that the x-axis points along the path. Thus, if you use the `transform shape` option, the node will "point along" the path.
9. In case a node is at `pos=0` or at `pos=1` some special code will suppress the superfluous lines to the start or end of the path.

The net effect of all of the above is that a node will be placed "on the path" and the path will have a "gap" just large enough to encompass the node. Another effect is that you can use this key multiple times on a path to add several node to a path, provided they do not overlap.



```
\usetikzlibrary {circuits}
\begin{tikzpicture}[circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,near start},
                  circuit handle symbol={draw,shape=circle,near end}] (3,2);
\end{tikzpicture}
```

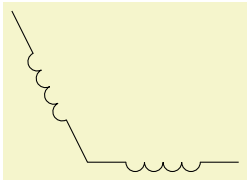


```
\usetikzlibrary {circuits}
\begin{tikzpicture}[transform shape,circuit]
  \draw (0,0) to [circuit handle symbol={draw,shape=rectangle,at start},
                  circuit handle symbol={draw,shape=circle,near end}] (3,2);
\end{tikzpicture}
```

49.2.3 Pointing Symbols in the Right Direction

Unlike normal nodes, which generally should not be rotated since this will make their text hard to read, symbols often need to be rotated. There are two ways of achieving such rotations:

1. When you place a symbol on a `to` path, the graphic symbol is automatically rotated such that it "points along the path". Here is an examples that shows how the inductor shape (which looks, unrotated, like this: ⌒⌒⌒) is automatically rotated around:



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \draw (3,0) to[inductor] (1,0) to[inductor] (0,2);
```

2. Many shapes cannot be placed "on" a path in this way, namely whenever there are more than two possible inputs. Also, you may wish to place the nodes first, possibly using a matrix, and connect them afterwards. In this case, you can simply add rotations like `rotate=90` to the shapes to rotate them. The following four keys make this slightly more convenient:

`/tikz/point up` (no value)

This is the same as `rotate=90`.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point up] {};
```

/tikz/point down (no value)

This is the same as rotate=-90.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point down] {};
```

/tikz/point left (no value)

This is the same as rotate=-180.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point left] {};
```

/tikz/point right (no value)

This key has no effect.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \node [diode,point right] {};
```

49.2.4 Info Labels

Info labels are used to add text to a circuit symbol. Unlike normal nodes like a rectangle, circuit symbols typically do not have text "on" them, but the text is placed next to them (like the text "3 Ω" next to a resistor).

TikZ already provides the label option for this purpose. The info option is built on top of this option, but it comes in some predefined variants that are especially useful in conjunction with circuits.

/tikz/info=[⟨*options*⟩]⟨*angle*⟩:⟨*text*⟩ (no default)

This key has nearly the same effect as the label key, only the following style is used additionally automatically:

/tikz/every info (style, no value)

Set this style to configure the styling of info labels. Since this key is *not* used with normal labels, it provides an easy way of changing the way info labels look without changing other labels.

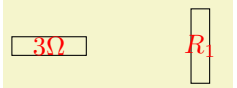
The ⟨*options*⟩ and ⟨*angle*⟩ are passed directly to the label command.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \node [resistor,info=$3\Omega$] {};
\end{tikzpicture}
```

You will find a detailed discussion of the label option on page 249.

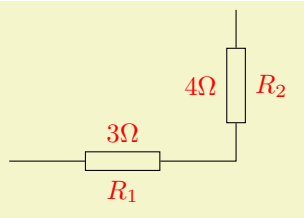
Hint: To place some text *on* the main node, use center as the ⟨*angle*⟩:




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \node [resistor,info=center:$3\Omega$] {};
  \node [resistor,point up,info=center:$R_1$] at (2,0) {};
\end{tikzpicture}
```

/tikz/info'=[⟨*options*⟩]⟨*angle*⟩:⟨*text*⟩ (no default)

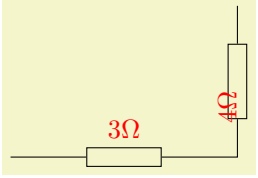
This key works exactly like the info key, only in case the ⟨*angle*⟩ is missing, it defaults to below instead of the current value of label position, which is usually above. This means that when you use info, you get a label above the node, while when you use the info' key you get a label below the node. In case the node has been rotated, the positions of the info nodes are rotated accordingly.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info={$3\Omega$},info'={$R_1$}}] (3,0)
              to[resistor={info={$4\Omega$},info'={$R_2$}}] (3,2);
\end{tikzpicture}
```

`/tikz/info sloped=`[⟨*options*⟩]⟨*angle*⟩`:`⟨*text*⟩ (no default)

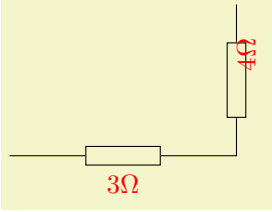
This key works like `info`, only the `transform shape` option is set when the label is drawn, causing it to follow the sloping of the main node.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info sloped={$3\Omega$}}] (3,0)
              to[resistor={info sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

`/tikz/info' sloped=` (no default)

This is a combination of `info'` and `info sloped`.



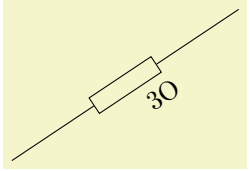

```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,every info/.style=red]
  \draw (0,0) to[resistor={info' sloped={$3\Omega$}}] (3,0)
              to[resistor={info' sloped={$4\Omega$}}] (3,2);
\end{tikzpicture}
```

`/tikz/circuit declare unit={`⟨*name*⟩`}{`⟨*unit*⟩`}` (no default)

This key is used to declare keys that make it easy to attach physical units to nodes. The idea is that instead of `info=$3\Omega$` you can write `ohm=3` or instead of `info'=$5\mathrm{S}$` you can write `siemens'=5`.

In detail, four keys are defined, namely `/tikz/`⟨*name*⟩, `/tikz/`⟨*name*⟩`'`, `/tikz/`⟨*name*⟩ `sloped`, and `/tikz/`⟨*name*⟩`' sloped`. The arguments of all of these keys are of the form [⟨*options*⟩]⟨*angle*⟩`:`⟨*value*⟩ and it is passed (slightly modified) to the corresponding key `info`, `info'`, `info sloped`, or `info' sloped`. The "slight modification" is the following: The text that is passed to the, say, `info` key is not ⟨*value*⟩, but rather `$\mathrm{`⟨*value*⟩⟨*unit*⟩`}$`

This means that after you said `circuit declare unit={ohm}{\Omega}`, then `ohm=5k` will have the same effect as `info={[every ohm]$\mathrm{5k\Omega}$}`. Here, `every ohm` is a style that allows you to configure the appearance of this unit. Since the `info` key is used internally, by changing the `every info` style, you can change the appearance of all units infos.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC,circuit declare unit={my ohm}{O}]
  \draw (0,0) to[resistor={my ohm' sloped=3}] (3,2);
\end{tikzpicture}
```

49.2.5 Declaring and Using Annotations

Annotations are quite similar to info labels. The main difference is that they generally cause something to be drawn by default rather than some text to be added (although an annotation might also add some text).

Annotations can be declared using the following key:

`/tikz/circuit declare annotation={`⟨*name*⟩`}{`⟨*distance*⟩`}{`⟨*path*⟩`}` (no default)

This key is used to declare an annotation named ⟨*name*⟩. Once declared, it can be used as an argument of a symbol and will add the drawing in ⟨*path*⟩ to the symbol. In detail, the following happens:

The Main Keys. Two keys called ⟨*name*⟩ and ⟨*name*⟩`'` are defined. The second causes the annotation to be "mirrored and placed on the other side" of the symbol. Both of these keys may also take further keys as parameter like `info` keys. Whenever the ⟨*name*⟩ key is used, a local scope is opened and in this scope the following things are done:

1. The style `every` ⟨*name*⟩ is executed.
2. The following style is executed and then `arrows=->`:

   `/tikz/annotation arrow` (style, no value)

   This style should set the `>` key to some desirable arrow tip.
3. The coordinate system is shifted such that the origin is at the north anchor of the symbol. (For the ⟨*name*⟩`'` key the coordinate system is flipped and shifted such that the origin is at the south anchor of the symbol.)
4. The `label distance` is locally set to ⟨*distance*⟩.
5. The parameter options given to the ⟨*name*⟩ key are executed.
6. The ⟨*path*⟩ is executed.

Usage. What all of the above amounts to is best explained by an example. Suppose we wish to create an annotation that looks like a little circular arrow (like ↻). We could then say:

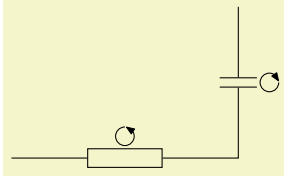
```
\tikzset{circuit declare annotation=
  {circular annotation}
  {9pt}
  {(0pt,8pt) arc (-270:80:3.5pt)}
}
```

We can then use it like this:



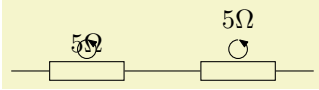
```
{\usetikzlibrary {circuits.ee.IEC}} pre
\tikz[circuit ee IEC]
  \draw (0,0) to [resistor={circular annotation}]   (3,0);
```

Well, not very impressive since we do not see anything. This is due to the fact that the ⟨*path*⟩ becomes part of a path that contains the symbol node an nothing else. This path is not drawn or filled, so we do not see anything. What we must do is to use an `edge` path operation:



```
\usetikzlibrary {circuits.ee.IEC}
\tikzset{circuit declare annotation={circular annotation}{9pt}
  {(0pt,8pt) edge[to path={arc(-270:80:3.5pt)}] ()}
}
\tikz[circuit ee IEC]
  \draw (0,0) to [resistor={circular annotation}]   (3,0)
              to [capacitor={circular annotation'}] (3,2);
```

The ⟨*distance*⟩ is important for the correct placement of additional `info` labels. When an annotation is present, the info labels may need to be moved further away from the symbol, but not always. For this reason, an annotation defines an additional ⟨*distance*⟩ that is applied to all info labels given as parameters to the annotation. Here is an example, that shows the difference:

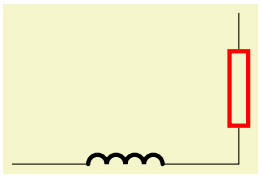



```
{\usetikzlibrary {circuits.ee.IEC}} pre
\tikz[circuit ee IEC]
  \draw (0,0) to [resistor={circular annotation,ohm=5}]   (2,0)
              to [resistor={circular annotation={ohm=5}}] (4,0);
```

49.2.6 Theming Symbols

For each symbol, a certain graphical representation is chosen to actually show the symbol. You can modify this graphical representation in several ways:

- You can select a different library and use a different `circuit ...` key. This will change all graphics used for the symbols.
- You can generally change the size of graphic symbols by setting `circuit size unit` to a different value or using a key like `small circuit symbols`.
- You can add options to the graphics used by symbols either globally by setting the `every circuit symbol` style or locally by setting the `every` ⟨*name*⟩ style, where ⟨*name*⟩ is the name of a symbol. For instance, in the following picture the symbols are ridiculously thick and resistors are red.



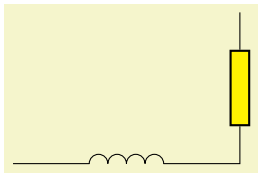
```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  [circuit ee IEC,
   every circuit symbol/.style={ultra thick},
   every resistor/.style={red}]

  \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
\end{tikzpicture}
```

- You can selectively change the graphic used for a symbol by saying `set resistor graphic=`.
- You can change one or more of the following styles:

`/tikz/circuit symbol open` (style, initially `draw`)

This style is used with symbols that consist of lines that surround some area. For instance, the IEC version of a resistor is an open symbol.



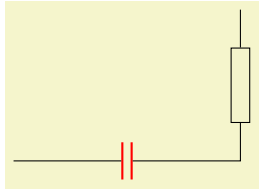
```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,
       circuit symbol open/.style={thick,draw,fill=yellow}]
  \draw (0,0) to [inductor] ++(right:3) to [resistor] ++(up:2);
```

`/tikz/circuit symbol filled` (style, initially `draw,fill=black`)

This style is used with symbols that are completely filled. For instance, the variant IEC version of an inductor is a filled, black rectangle.

`/tikz/circuit symbol lines` (style, initially `draw`)

This style is used with symbols that consist only of lines that do not surround anything. Examples are a capacitor.

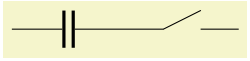


```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,
       circuit symbol lines/.style={thick,draw=red}]
  \draw (0,0) to [capacitor] ++(right:3) to [resistor] ++(up:2);
```

`/tikz/circuit symbol wires` (style, initially `draw`)

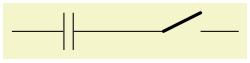
This style is used for symbols that consist only of "wires". The difference to the previous style is that a symbol consisting of wires will look strange when the lines are thicker than the lines of normal wires, while for symbols consisting of lines (but not wires) it may look nice to make them thicker. An example is the `make contact` symbol.

Compare



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,circuit symbol lines/.style={draw,very thick}]
  \draw (0,0) to [capacitor={near start},
                  make contact={near end}] (3,0);
```

to



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC,circuit symbol wires/.style={draw,very thick}]
  \draw (0,0) to [capacitor={near start},
                  make contact={near end}] (3,0);
```

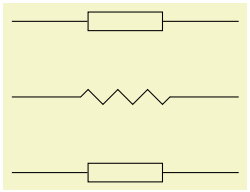
All circuit environments like `circuit logic IEC` mainly use options like `set and gate graphic=...` to set up the graphics used for a certain symbol. It turns out that graphic hidden in the "`...`" part is also always available as a separate style, whose name contains the library's initials. For instance, the `circuit logic IEC` option actually contains the following command:

```
  set and gate graphic = and gate IEC graphic,
```

The `and gate IEC graphic` style, in turn, is defined as follows:

```
\tikzset{and gate IEC graphic/.style=
  {
    circuit symbol open,
    circuit symbol size=width 2.5 height 4,
    shape=and gate IEC,
    inner sep=.5ex
  }
}
```

Normally, you do not need to worry about this, since you will not need to access a style like `and gate IEC graphic` directly; you will only use the `and gate` key. However, sometimes libraries define *variants* of a graphic; for instance, there are two variants for the resistor graphic in the IEC library. In this case you can set the graphic for the resistor to this variant (or back to the original) by saying `set resistor graphic` yourself:



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[circuit ee IEC]
  % Standard resistor
  \draw (0,2) to [resistor] (3,2);

  % Var resistor
  \begin{scope}[set resistor graphic=var resistor IEC graphic]
    \draw (0,1) to [resistor] (3,1);

    % Back to original
    \draw [set resistor graphic=resistor IEC graphic]
      (0,0) to [resistor] (3,0);
  \end{scope}
\end{tikzpicture}
```

49.3 Logical Circuits

49.3.1 Overview

A *logical circuit* is a circuit that contains what we call *logical gates* like an `and gate` or an `xor gate`. The logical libraries are intended to make it easy to draw such circuits.

In the following, we first have a look at the different libraries that can be used in principle and how the symbols look like. Then we have a more detailed look at how the symbols are used. Finally, we discuss the implementation details.

There are different ways of depicting logical gates, which is why there are different (sub-)libraries for drawing them. They provide the necessary graphical representations of the symbols declared in the following library:

TikZ Library `circuits.logic`

```
\usetikzlibrary{circuits.logic} % LaTeX and plain TeX
\usetikzlibrary[circuits.logic] % ConTeXt
```

This library declares the logical gate symbols, but does not provide the symbol graphics. The library also defines the following key which, however, is also only used indirectly, namely by other libraries:

/tikz/circuit logic (no value)

This style calls the keys `circuit` (which internally calls `every circuit`, then it defines the `inputs` key and it calls the `every circuit logic` key.

/tikz/inputs=⟨*inputs*⟩ (no default)

This key is defined only inside the scope of a `circuit logic`. There, it has the same effect as `logic gate inputs`, described on page 621.

/tikz/every circuit logic (style, no value)

Use this key to configure the appearance of logical circuits.

Since the `circuits.logic` library does not define any actual graphics, you need to use one of the following libraries, instead:

TikZ Library circuits.logic.IEC

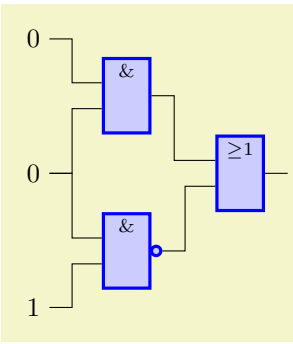
```
\usetikzlibrary{circuits.logic.IEC} % LaTeX and plain TeX
\usetikzlibrary[circuits.logic.IEC] % ConTeXt
```

This library provides graphics based on gates recommended by the International Electrotechnical Commission. When you include this library, you can use the following key to set up a scope that contains a logical circuit where the gates are shown in this style.

/tikz/circuit logic IEC (no value)

This key calls `circuit logic` and installs the IEC-like graphics for the logical symbols like `and gate`.

As explained in Section 49.2.6, for each graphic symbol of the library there is also a style that stores this particular appearance. These keys are called `and gate IEC graphic`, `or gate IEC graphic`, and so on.




```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC,
                    every circuit symbol/.style={
                      logic gate IEC symbol color=black,
                      fill=blue!20,draw=blue,very thick}]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                              & \\
                    & \node [and gate] (a1) {};    & \\
    \node (i1) {0}; &                              & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {};   & \\
    \node (i2) {1}; &                              & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

TikZ Library circuits.logic.US

```
\usetikzlibrary{circuits.logic.US} % LaTeX and plain TeX
\usetikzlibrary[circuits.logic.US] % ConTeXt
```

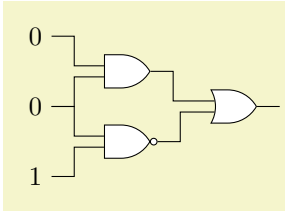
This library provides graphics showing "American" logic gates. It defines the following key:

/tikz/circuit logic US (no value)

This style calls `circuit logic` and installs US-like graphics for the logical symbols like `and gate`. For instance, it says

```
set and gate graphic = and gate US graphic
```

Here is an example:




```
\usetikzlibrary {circuits.logic.CDH}
\begin{tikzpicture}[circuit logic CDH,
                    tiny circuit symbols,
                    every circuit symbol/.style={
                      fill=white,draw}]
  \matrix[column sep=7mm]
  {
    \node (i0) {0}; &                              & \\
                    & \node [and gate] (a1) {};    & \\
    \node (i1) {0}; &                              & \node [or gate] (o) {};\\
                    & \node [nand gate] (a2) {};   & \\
    \node (i2) {1}; &                              & \\
  };
  \draw (i0.east) -- ++(right:3mm) |- (a1.input 1);
  \draw (i1.east) -- ++(right:3mm) |- (a1.input 2);
  \draw (i1.east) -- ++(right:3mm) |- (a2.input 1);
  \draw (i2.east) -- ++(right:3mm) |- (a2.input 2);
  \draw (a1.output) -- ++(right:3mm) |- (o.input 1);
  \draw (a2.output) -- ++(right:3mm) |- (o.input 2);
  \draw (o.output) -- ++(right:3mm);
\end{tikzpicture}
```

TikZ Library circuits.logic.CDH

```
\usetikzlibrary{circuits.logic.CDH} % LaTeX and plain TeX
\usetikzlibrary[circuits.logic.CDH] % ConTeXt
```

This library provides graphics based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. They are identical to the US-style symbols, except for the and- and nand-gates.

/tikz/circuit logic CDH (no value)

This key calls `circuit logic US` and installs the two special and- and nand-gates, that is, it uses `set and gate graphic` with `and gate CDH graphic` and likewise for nand-gates.

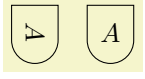
Inside `circuit logic XYZ` scopes, you can now use the keys shown in Section 49.3.2. We have a more detailed look at one of them, all the other work the same way:

/tikz/and gate (no value)

This key should be passed to a `node` command. It will cause the node to "look like" an `and gate`, where the exact appearance of the gate is dictated by the which circuit environment is used. To further configure the appearance of the `and gate`, see Section 49.2.6.




```
\usetikzlibrary {circuits.logic.IEC}
\tikz [circuit logic IEC] \node [and gate] {$A$};
```




```
\usetikzlibrary {circuits.logic.US}
\tikz [circuit logic US]
{
  \node [and gate,point down] {$A$};
  \node [and gate,point down,info=center:$A$] at (1,0) {};
}
```

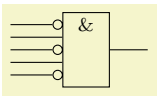
Inputs. Multiple inputs can be specified for a logic gate (provided they support multiple inputs: a not gate – also known as an inverter – does not). However, there is an upper limit for the number of inputs which has been set to 1024, which should be *way* more than would ever be needed.

The following key is used to configure the inputs. It is available only inside a `circuit logic` environment.

/tikz/inputs=⟨*input list*⟩ (no default, initially `{normal,normal}`)

If a gate has n inputs, the ⟨*input list*⟩ should consists of n letters, each being `i` for "inverted" or `n` for "normal". Inverted gates will be indicated by a little circle. In any case the anchors for the

inputs will be set up appropriately, numbered from top to bottom `input 1`, `input 2`, ...and so on. If the gate only supports one input the anchor is simply called `input` with no numerical index.




```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[circuit logic IEC]
  \node[and gate,inputs={inini}] (A) {};
  \foreach \a in {1,...,5}
    \draw (A.input \a -| -1,0) -- (A.input \a);
  \draw (A.output) -- ++(right:5mm);
\end{tikzpicture}
```

(This key is just a shorthand for `logic gate inputs`, described in detail on page 621. There you will also find descriptions of how to configure the size of the inverted circles and the way the symbol size increases when there are too many inputs.)

Output. Every logic gate has one anchor called `output`.

49.3.2 Symbols: The Gates

The following table shows which symbols are declared by the main `circuits.logic` library and their appearance in the different sublibraries.

Key	*Appearance inside* `circuit logic IEC`	*Appearance inside* `circuit logic US`	*Appearance inside* `circuit logic CDH`
`/tikz/and gate`			
`/tikz/nand gate`			
`/tikz/or gate`			
`/tikz/nor gate`			
`/tikz/xor gate`			
`/tikz/xnor gate`			
`/tikz/not gate`			
`/tikz/buffer gate`			

49.3.3 Implementation: The Logic Gates Shape Library

The previous sections described the Ti*k*Z interface for creating logical circuits. In this section we take a closer look at the underlying PGF libraries.

Just as there are several Ti*k*Z circuit libraries, there are two underlying PGF shape libraries, one for creating US-style gates and one for IEC-style gates. These libraries define *shapes* only. It is the job of the circuit libraries to "theme" them so that they "look nice". However, in principle, you can also use these shapes directly.

Let us begin with the base library that defines the handling of inputs.

Ti*k*Z Library `shapes.gates.logic`

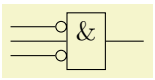
```
\usepgflibrary{shapes.gates.logic}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.gates.logic]  % ConTeXt when using pure PGF
```

```
\usetikzlibrary{shapes.gates.logic} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic] % ConTeXt when using TikZ
```

This library defines common keys used by all logical gate shapes.

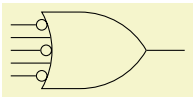
`/pgf/logic gate inputs=`⟨*input list*⟩ (no default, initially `{normal,normal}`)

Specify the inputs for the logic gate. The keyword `inverted` indicates an inverted input which will mean PGF will draw a circle attached to the main shape of the logic gate. Any keyword that is not `inverted` will be treated as a "normal" or "non-inverted" input (however, for readability, you may wish to use `normal` or `non-inverted`), and PGF will not draw the circle. In both cases the anchors for the inputs will be set up appropriately, numbered from top to bottom `input 1`, `input 2`, …and so on. If the gate only supports one input the anchor is simply called `input` with no numerical index.




```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[minimum height=0.75cm]
  \node[and gate IEC, draw, logic gate inputs={inverted, normal, inverted}]
    (A) {};
  \foreach \a in {1,...,3}
    \draw (A.input \a -| -1,0) -- (A.input \a);
  \draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

For multiple inputs it may be somewhat unwieldy to specify a long list, thus, the following "shorthand" is permitted (this is an extension of ideas due to Jürgen Werber and Christoph Bartoschek): Using `i` for inverted and `n` for normal inputs, ⟨*input list*⟩ can be specified *without the commas*. So, for example, `ini` is equivalent to `inverted, normal, inverted`.



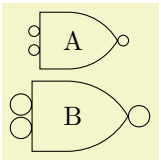
```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture}[minimum height=0.75cm]
  \node[or gate US, draw,logic gate inputs=inini] (A) {};
  \foreach \a in {1,...,5}
    \draw (A.input \a -| -1,0) -- (A.input \a);
  \draw (A.output) -- ([xshift=0.5cm]A.output);
\end{tikzpicture}
```

The height of the gate may be increased to accommodate the number of inputs. In fact, it depends on three variables: n, the number of inputs, r, the radius of the circle used to indicate an inverted input and s, the distance between the centers of the inputs. The default height is then calculated according to the expression $(n+1) \times \max(2r, s)$. This then may be increased to accommodate the node contents or any minimum size specifications.

The radius of the inverted input circle and the distance between the centers of the inputs can be customized using the following keys:

`/pgf/logic gate inverted radius=`⟨*length*⟩ (no default, initially `2pt`)

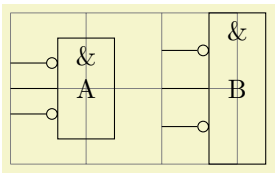
Set the radius of the circle that is used to indicate inverted inputs. This is also the radius of the circle used for the inverted output of the `nand`, `nor`, `xnor` and `not` gates.




```
\usetikzlibrary {circuits.logic.CDH}
\begin{tikzpicture}[minimum height=0.75cm]
  \tikzset{every node/.style={shape=nand gate CDH, draw, logic gate inputs=ii}}
  \node[logic gate inverted radius=2pt] {A};
  \node[logic gate inverted radius=4pt] at (0,-1) {B};
\end{tikzpicture}
```

`/pgf/logic gate input sep=`⟨*length*⟩ (no default, initially `.125cm`)

Set the distance between the *centers* of the inputs to the logic gate.




```
\usetikzlibrary {circuits.logic.IEC}
\begin{tikzpicture}[minimum size=0.75cm]
  \draw [help lines] grid (3,2);
  \tikzset{every node/.style={shape=and gate IEC, draw, logic gate inputs=ini}}
  \node[logic gate input sep=0.33333cm] at (1,1)(A) {A};
  \node[logic gate input sep=0.5cm]     at (3,1) (B) {B};
  \foreach \a in {1,...,3}
    \draw (A.input \a -| 0,0) -- (A.input \a)
          (B.input \a -| 2,0) -- (B.input \a);
\end{tikzpicture}
```

PGF will increase the size of the logic gate to accommodate the number of inputs, and the size of the inverted radius and the separation between the inputs. However with all shapes in this library, any increase in size (including any minimum size requirements) will be applied so that the default aspect ratio is unaltered. This means that changing the height will change the width and vice versa.

49.3.4 Implementation: The US-Style Logic Gates Shape Library

TikZ Library shapes.gates.logic.US

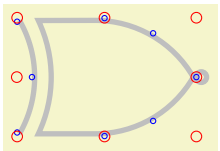
```
\usepgflibrary{shapes.gates.logic.US}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.gates.logic.US]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.gates.logic.US} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic.US] % ConTeXt when using TikZ
```

This library provides "American" logic gate shapes whose names are suffixed with the identifier `US`. Additionally, alternative `and` and `nand` gates are provided which are based on the logic symbols used in A. Croft, R. Davidson, and M. Hargreaves (1992), *Engineering Mathematics*, Addison-Wesley, 82–95. These two shapes are suffixed with `CDH`.

The "compass point" anchors apply to the main part of the shape and do not include any inverted inputs or outputs. This library provides an additional feature to facilitate the relative positioning of logic gates:

`/pgf/logic gate anchors use bounding box=`⟨*boolean*⟩ (no default, initially `false`)

When set to `true` this key will ensure that the compass point anchors use the bounding rectangle of the main shape, which, ignore any inverted inputs or outputs, but includes any `outer sep`. This *only* affects the compass point anchors and is not set on a shape by shape basis: whether the bounding box is used is determined by value of this key when the anchor is accessed.



```
\usetikzlibrary {circuits.logic.US}
\begin{tikzpicture}[minimum height=1.5cm]
  \node[xnor gate US, draw, gray!50,line width=2pt] (A) {};
  \foreach \x/\y/\z in {false/blue/1pt, true/red/2pt}
    \foreach \a in {north, south, east, west, north east,
      south east, north west, south west}
      \draw[logic gate anchors use bounding box=\x, color=\y]
        (A.\a) circle[radius=\z];
\end{tikzpicture}
```

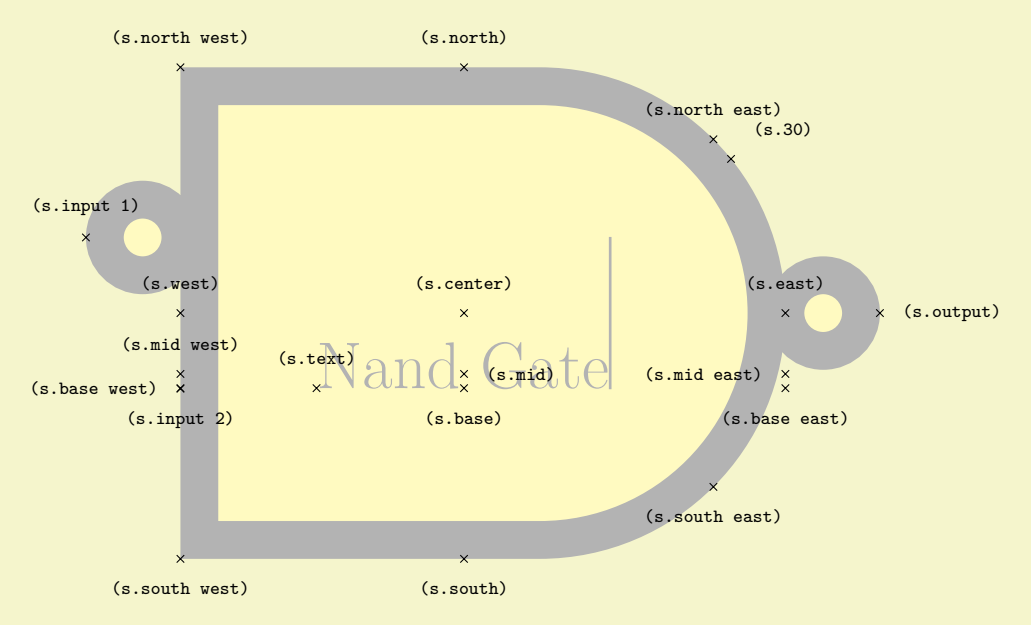
The library defines a number of shapes. For each shape the allowed number of inputs is also shown:

- `and gate US`, two or more inputs
- `and gate CDH`, two or more inputs
- `nand gate US`, two or more inputs
- `nand gate CDH`, two or more inputs
- `or gate US`, two or more inputs
- `nor gate US`, two or more inputs
- `xor gate US`, two or more inputs
- `xnor gate US`, two or more inputs
- `not gate US`, one input
- `buffer gate US`, one input

In the following, we only have a detailed look at the anchors defined by one of them. We choose the `nand gate US` because it shows all the "interesting" anchors.

Shape `nand gate US`

This shape is a nand gate, which supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two non-inverted inputs (using the normal compass point anchors) are shown below. Anchor `30` is an example of a border anchor.




```
\usetikzlibrary {circuits.logic.US}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=nand gate US,shape example, inner sep=0cm,
  logic gate inputs={in},
  logic gate inverted radius=.5cm] {Nand Gate\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {center/above, text/above, 30/above right,
     mid/right, mid east/left, mid west/above,
     base/below, base east/below, base west/left,
     north/above, south/below, east/above, west/above,
     north east/above, south east/below, south west/below, north west/above,
     output/right, input 1/above, input 2/below}
     \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

(For the definition of the `shape example` style, see Section 71.)

49.3.5 Implementation: The IEC-Style Logic Gates Shape Library

Ti*k*Z Library `shapes.gates.logic.IEC`

```
\usepgflibrary{shapes.gates.logic.IEC}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.gates.logic.IEC]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.gates.logic.IEC} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.logic.IEC] % ConTeXt when using TikZ
```

This library provides rectangular logic gate shapes. These shapes are suffixed with `IEC` as they are based on gates recommended by the International Electrotechnical Commission.

By default each gate is drawn with a symbol, & for `and` and `nand` gates, ≥ 1 for `or` and `nor` gates, 1 for `not` and `buffer` gates, and $= 1$ for `xor` and `xnor` gates. These symbols are drawn automatically (internally they are drawn using the "foreground" path), and are not strictly speaking part of the node contents. However, the gate is enlarged to make sure the symbols are within the border of the node. It is possible to change the symbols and their position within the node using the following keys:

/pgf/and gate IEC symbol=⟨*text*⟩ (no default, initially `\char`\&`)

Set the symbol for the `and gate`. Note that if the node is filled, this color will be used for the symbol, making it invisible, so it will be necessary set ⟨*text*⟩ to something like `\color{black}\char`\&`. Alternatively, the `logic gate IEC symbol color` key can be used to set the color of all symbols simultaneously.

In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `and gate symbol`.

/pgf/nand gate IEC symbol=⟨*text*⟩ (no default, initially `\char`\&`)

Set the symbol for the `nand gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `nand gate symbol`.

/pgf/or gate IEC symbol=⟨*text*⟩ (no default, initially `$\geq1$`)

Set the symbol for the `or gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `or gate symbol`.

/pgf/nor gate IEC symbol=⟨*text*⟩ (no default, initially `$\geq1$`)

Set the symbol for the `nor gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `nor gate symbol`.

/pgf/xor gate IEC symbol=⟨*text*⟩ (no default, initially `{$=1$}`)

Set the symbol for the `xor gate`. Note the necessity for braces, as the symbol contains `=`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `xor gate symbol`.

/pgf/xnor gate IEC symbol=⟨*text*⟩ (no default, initially `{$=1$}`)

Set the symbol for the `xnor gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `xnor gate symbol`.

/pgf/not gate IEC symbol=⟨*text*⟩ (no default, initially `1`)

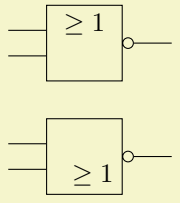
Set the symbol for the `not gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `not gate symbol`.

/pgf/buffer gate IEC symbol=⟨*text*⟩ (no default, initially `1`)

Set the symbol for the `buffer gate`. In Ti*k*Z, when the `use IEC style logic gates` key has been used, this key can be replaced by `buffer gate symbol`.

/pgf/logic gate IEC symbol align=⟨*align*⟩ (no default, initially `top`)

Set the alignment of the logic gate symbol (in Ti*k*Z, when the `use IEC style logic gates` key has been used, `IEC` can be omitted). The specification in ⟨*align*⟩ is a comma separated list from `top`, `bottom`, `left` or `right`. The distance between the border of the node and the outer edge of the symbol is determined by the values of the `inner xsep` and `inner ysep`.




```
\usetikzlibrary {shapes.gates.logic.IEC}
\begin{tikzpicture}[minimum size=1cm, use IEC style logic gates]
    \tikzset{every node/.style={nor gate, draw}}
  \node (A) at (0,1.5) {};
  \node [logic gate symbol align={bottom, right}] (B) at (0,0) {};
  \foreach \g in {A, B}{
    \foreach \i in {1,2}
      \draw ([xshift=-0.5cm]\g.input \i) -- (\g.input \i);
    \draw (\g.output) -- ([xshift=0.5cm]\g.output);
  }
\end{tikzpicture}
```

/pgf/logic gate IEC symbol color=⟨*color*⟩ (no default)

This key sets the color for all symbols simultaneously. This color can be overridden on a case by case basis by specifying a color when setting the symbol text.

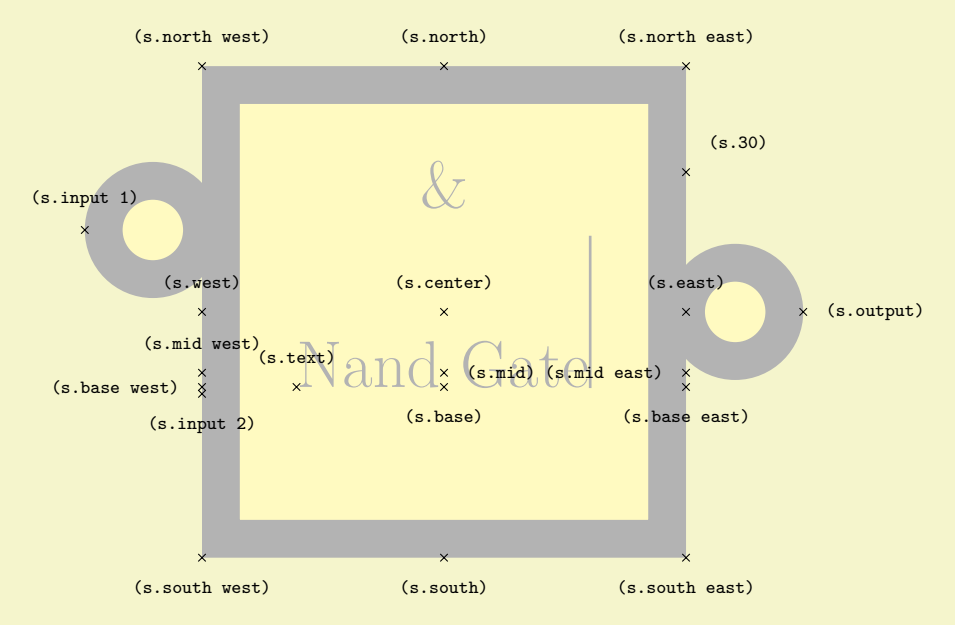
The library defines the following shapes:

- `and gate IEC`, two or more inputs
- `nand gate IEC`, two or more inputs
- `or gate IEC`, two or more inputs
- `nor gate IEC`, two or more inputs
- `xor gate IEC`, two or more inputs
- `xnor gate IEC`, two or more inputs
- `not gate IEC`, one input
- `buffer gate IEC`, one input

Again, we only have a look at the nand-gate in more detail:

Shape `nand gate IEC`

This shape is a nand gate. It supports two or more inputs. If less than two inputs are specified an error will result. The anchors for this gate with two inverted inputs are shown below. Anchor `30` is an example of a border anchor.




```
\usetikzlibrary {circuits.logic.IEC}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=nand gate IEC ,shape example, inner xsep=1cm, inner ysep=1cm,
    minimum height=6cm, nand gate IEC symbol=\color{black!30}\char`\&,
    logic gate inputs={in},
    logic gate inverted radius=0.65cm]
  {Nand Gate\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {center/above, text/above, 30/above right,
     mid/right, mid east/left, mid west/above,
     base/below, base east/below, base west/left,
     north/above, south/below, east/above, west/above,
     north east/above, south east/below, south west/below, north west/above,
     output/right, input 1/above, input 2/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.4 Electrical Engineering Circuits

49.4.1 Overview

An *electrical engineering circuit* contains symbols like resistors or capacitors or voltage sources and annotations like the two arrows pointing toward an element whose behaviour is light dependent. The electrical

engineering libraries, abbreviated ee-libraries, provide such symbols and annotations.

Just as for logical gates, there are different ways of drawing ee-symbols. Currently, there is one main library for drawing circuits, which uses the graphics from the International Electrotechnical Commission, but you can add your own libs. This is why, just as for logical gates, there is a base library and more specific libraries.

TikZ Library circuits.ee

```
\usetikzlibrary{circuits.ee} % LaTeX and plain TeX
\usetikzlibrary[circuits.ee] % ConTeXt
```

This library declares the ee symbols, but (mostly) does not provide the symbol graphics, which is left to the sublibraries. Just like the logical gates library, a key is defined that is normally only used internally:

/tikz/circuit ee (no value)

This style calls the keys `circuit` (which internally calls `every circuit` and the following style:

/tikz/every circuit ee (style, no value)

Use this key to configure the appearance of logical circuits.

The library also declares some standard annotations and units.

As for logical circuits, to draw a circuit the first step is to include a library containing the symbols graphics. Currently, you have to include `circuits.ee.IEC`.

TikZ Library circuits.ee.IEC

```
\usetikzlibrary{circuits.ee.IEC} % LaTeX and plain TeX
\usetikzlibrary[circuits.ee.IEC] % ConTeXt
```

When this library is loaded, you can use the following style:

/tikz/circuit ee IEC (no value)

This style calls `circuit ee` and installs the IEC-like graphics for the logical symbols like `resistor`.

Inside the `circuit ee IEC` scope, you can now use the keys for symbols, units, and annotations listed in the later sections. We have a more detailed look at one of each of them, all the others work the same way.

Let us start with an example of a symbol: the resistor symbol. The other predefined symbols are listed in Section 49.4.2 and later sections.

/tikz/resistor=⟨*options*⟩ (no default)

This key should be used with a `node` path command or with the `to` path command.

Using the Key with Normal Nodes. When used with a node, it will cause this node to "look like" a resistor (by default, in the IEC library, this is just a simple rectangle).



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \node [resistor] {};
```

Unlike normal nodes, a resistor node generally should not take any text (as in `node [resistor] {foo}`). Instead, the labeling of resistors should be done using the `label`, `info` and `ohm` options.



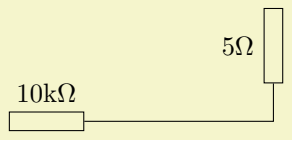

```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \node [resistor,ohm=5] {};
```

The ⟨*options*⟩ make no real sense when the `resistor` option is used with a normal node, you can just as well given them to the `node` itself. Thus, the following has the same effect as the above example:




```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \node [resistor={ohm=5}] {};
```

In a circuit, you will often wish to rotate elements. For this, the options `point up`, `point down`, `point left` or `point right` may be especially useful. They are just shorthands for appropriate rotations like `rotate=90`.

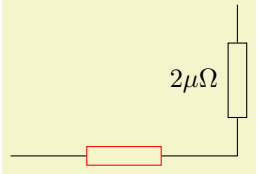



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] {
  \node (R1) [resistor,point up,ohm=5] at (3,1) {};
  \node (R2) [resistor,ohm=10k]        at (0,0) {};
  \draw (R2) -| (R1);
}
```

Using the Key on a To Path. When the `resistor` key is used on a `to` path inside a `circuit ee IEC`, the `circuit handle symbol` key is called internally. This has a whole bunch of effects:

1. The path currently being constructed is cut up to make place for a node.
2. This node will be a `resistor node` that is rotated so that it points "along" the path (unless an option like `shift only` or an extra rotation is used to change this).
3. The ⟨*options*⟩ passed to the `resistor` key are passed on to the node.
4. The ⟨*options*⟩ are pre-parsed to identify a `pos` key or a key like `at start` or `midway`. These keys are used to determine where on the `to` path the node will lie.

Since the ⟨*options*⟩ of the `resistor` key are passed on to the resistor node on the path, you can use it to add labels to the node. Here is a simple example:




```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \draw (0,0) to [resistor=red]         (3,0)
              to [resistor={ohm=2\mu}] (3,2);
```

You can add multiple labels to a resistor and you can have multiple resistors (or other elements) on a single path.

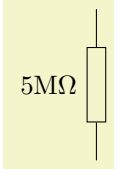
Inputs, Outputs, and Anchors. Like the logical gates, all ee-symbols have an `input` and an `output` anchor. Special-purpose-nodes may have even more anchors of this type. Furthermore, the ee-symbols-nodes also have four standard compass direction anchors.

Changing the Appearance. To configure the appearance of all `resistors`, see Section 49.2.6. You can use the ⟨*options*⟩ to locally change the appearance of a single resistor.

Let us now have a look at an example of a unit: the Ohm unit. The other predefined units are listed in Section 49.4.7.

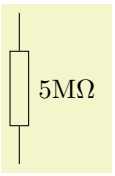
`/tikz/ohm`=⟨*value*⟩ (no default)

This key is used to add an `info` label to a node with a special text: `$\mathrm{`⟨*value*⟩`\Omega}$`. In other words, the `ohm` key can only be used with the options of a node and, when used, it will cause the ⟨*value*⟩ to be placed next to the node, followed by Ω. Since the ⟨*value*⟩ is typeset inside a `\mathrm` command, when you write `ohm=5k` you get 5kΩ, `ohm=5p` yields 5pΩ, and `ohm=5.6\cdot 10^{2}\mu` yields $5.6 \cdot 10^2 \mu\Omega$.



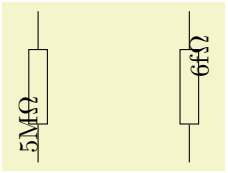

```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \draw (0,0) to [resistor={ohm=5M}] (0,2);
```

Instead of `ohm` you can also use `ohm'`, which places the label on the other side.




```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \draw (0,0) to [resistor={ohm'=5M}] (0,2);
```

Finally, there are also keys `ohm sloped` and `ohm' sloped` for having the info label rotate together with the main node.



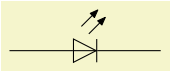

```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
  \draw (0,0) to [resistor={ohm sloped=5M}] (0,2)
        (2,0) to [resistor={ohm' sloped=6f}]  (2,2);
```

You can configure the appearance of an Ohm info label using the key `every ohm`.

Finally, let us have a look at an annotation: the `light emitting` annotation. The other predefined units are listed in Section 49.4.8.

`/tikz/light emitting=`⟨*options*⟩ (no default)

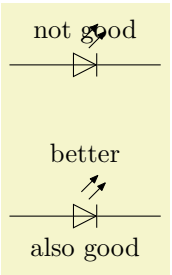
Like a unit, an annotation should be given as an additional option to a node. It causes some drawings (in this case, two parallel lines) to be placed next to the node.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC] \draw (0,0) to [diode=light emitting] (2,0);
```

The ⟨*options*⟩ can be used for three different things:

1. You can use keys like `red` to change the appearance of this annotation, locally.
2. You can use keys like `<-` or `-latex` to change the direction and kinds of arrows used in the annotation.
3. You can use info labels like `ohm=5` or `info=foo` inside the ⟨*options*⟩. These info labels will be added to the main node (not to the annotation itself), but the label distance will have been changed to accommodate for the space taken up by the annotation.




```
\usetikzlibrary {circuits.ee.IEC}
\tikz [circuit ee IEC]
{
  \draw (0,2) to [diode={light emitting,info=not good}] (2,2);
  \draw (0,0) to [diode={light emitting={info=better},
                         info'=also good}]  (2,0);
}
```

In addition to `light emitting` there is also a key called `light emitting'`, which simply places the annotation on the other side of the node.

You can configure the appearance of annotations in three ways:

- You can set the `every circuit annotation` style.
- You can set the `every light emitting` style.
- You can set the following key:

  `/tikz/annotation arrow` (style, no value)

  This style should set the default `>` arrow to some nice value.

49.4.2 Symbols: Indicating Current Directions

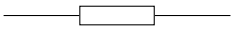
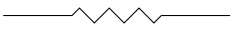
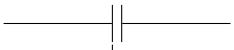
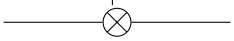
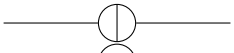
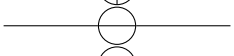
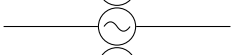
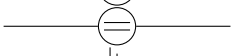
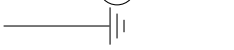
There are two symbols for indicating current directions. These symbols are defined directly inside `circuit ee`.

Key	*Appearance*
`/tikz/current direction`	
`/tikz/current direction'`	

The examples have been produced by (in essence) `\draw (0,0) to[`⟨*symbol name*⟩`] (3,0);`.

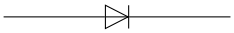
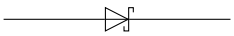
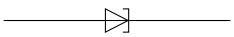
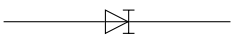
49.4.3 Symbols: Basic Elements

The following table show basic symbols as they are depicted inside the `circuit ee IEC` environment. To install one of alternate graphics, you have to say `set` ⟨*symbol name*⟩ `graphic=var` ⟨*symbol name*⟩ `IEC graphic`.

Key	*Appearance*	*Alternate appearance*
`/tikz/resistor`		
`/tikz/inductor`		
`/tikz/capacitor`		
`/tikz/battery`		
`/tikz/bulb`		
`/tikz/current source`		
`/tikz/voltage source`		
`/tikz/ac source`		
`/tikz/dc source`		
`/tikz/ground`		

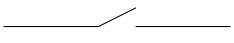
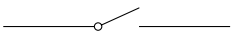
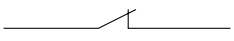
49.4.4 Symbols: Diodes

The following table shows diodes as they are depicted inside the `circuit ee IEC` environment.

Key	*Appearance*	*Alternate appearance*
`/tikz/diode`		
`/tikz/Zener diode`		
`/tikz/Schottky diode`		
`/tikz/tunnel diode`		
`/tikz/backward diode`		
`/tikz/breakdown diode`		

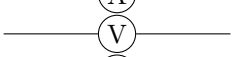
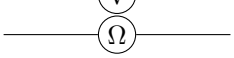
49.4.5 Symbols: Contacts

The following table shows contacts as they are depicted inside the `circuit ee IEC` environment.

Key	*Appearance*	*Alternate appearance*
`/tikz/contact`		
`/tikz/make contact`		
`/tikz/break contact`		

49.4.6 Symbols: Measurement devices

The following table shows measurement devices as they are depicted inside the `circuit ee IEC` environment.

Key	*Appearance*
`/tikz/amperemeter`	
`/tikz/voltmeter`	
`/tikz/ohmmeter`	

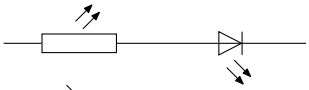
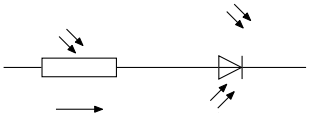
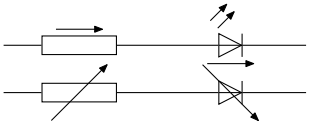
49.4.7 Units

The `circuits.ee` library predefines the following unit keys:

Key	*Appearance of* 1 *unit*
`/tikz/ampere`	1A
`/tikz/volt`	1V
`/tikz/ohm`	1Ω
`/tikz/siemens`	1S
`/tikz/henry`	1H
`/tikz/farad`	1F
`/tikz/coulomb`	1C
`/tikz/voltampere`	1VA
`/tikz/watt`	1W
`/tikz/hertz`	1Hz

49.4.8 Annotations

The `circuits.ee.IEC` library defines the following annotations:

Key	*Appearance*
`/tikz/light emitting`	
`/tikz/light dependent`	
`/tikz/direction info`	
`/tikz/adjustable`	

The lines have been produced using, in essence,

```
\draw (0,0)  to [resistor=light emitting] (2,0)  to [diode=light emitting'] (4,0);
```

and similarly for the other annotations.

49.4.9 Implementation: The EE-Symbols Shape Library

The Ti*k*Z libraries depend on two shape libraries, which are included automatically. Usually, you will not need to use these shapes directly.

Ti*k*Z Library `shapes.gates.ee`

```
\usepgflibrary{shapes.gates.ee}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.gates.ee]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.gates.ee} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.ee] % ConTeXt when using TikZ
```

This library defines basic shapes that can be used by all ee-circuit libraries. Currently, it defines the following shapes:

- `rectangle ee`
- `circle ee`
- `direction ee`

Additionally, the library defines the following arrow tip: The `direction ee` arrow tip is basically the same as a `triangle 45` arrow tip with rounded joins.

`direction ee` yields thick ⟷ and thin ⟷

However, unlike normal arrow tips, its size does *not* depend on the current line width. Rather, it depends on the value of its arrow options, which should be set to the desired size. Thus, you should say something like `\pgfsetarrowoptions{direction ee}{5pt}` to set the size of the arrow.

Shape `rectangle ee`

This shape is completely identical to a normal `rectangle`, only there are two additional anchors: The `input` anchor is an alias for the `west` anchor, while the `output` anchor is an alias for the `east` anchor.

Shape `circle ee`

Like the `rectangle ee` shape, only for circles.

Shape `direction ee`

This shape is rather special. It is intended to be used to "turn an arrow tip into a shape". First, you should set the following key to the name of an arrow tip:

`/pgf/direction ee arrow=`⟨*right arrow tip name*⟩ (no default)

The value of this key will be used for the arrow tip depicted in an `direction ee` shape.

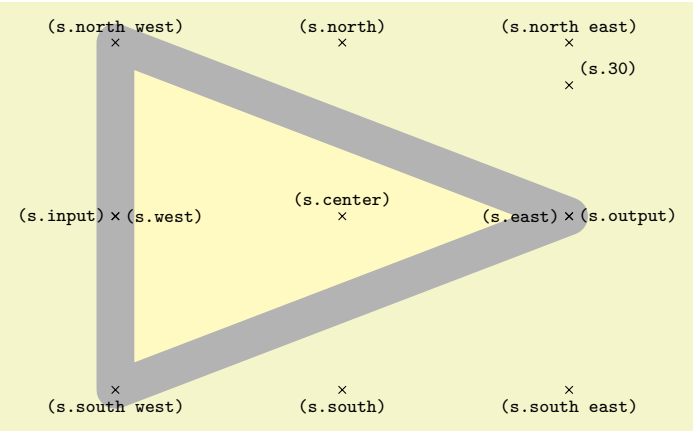
When a node of shape `direction ee` is created, several things happen:

1. The size of the shape is computed according to the following rules: The width of the shape is set up so that the left border of the shape is at the left end of the arrow tip and the right border is at the right end of the arrow tip. These left and right "ends" of the arrow are the tip end and the back end specified by the arrow itself (see Section 105.2 for details). You usually need not worry about this width setting.

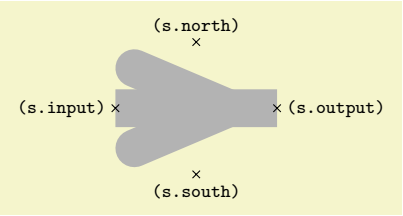
   By comparison, the height of the arrow is given by the current setting of `minimum height`. Thus, this key must have been set up correctly to reflect the "real" height of the arrow tip. The reason is that the height of an arrow is not specified when arrows are declared and is, thus, not available, here.

   Possibly, the height computation will change in the future to reflect the real height of the arrow, so you should generally set up the `minimum height` to be the same as the real height.
2. A straight line from left to right inside the shape's boundaries is added to the background path.
3. The arrow tip, pointing right, is drawn before the background path.

The anchors of this shape are just the compass anchors, which lie on a rectangle whose width and height are the above-computed height and width.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \pgfsetarrowoptions{direction ee}{6cm}
  \node[name=s,shape=direction ee,shape example,minimum height=0.7654*6cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}[direction ee arrow=angle 45]
  \node[name=s,shape=direction ee,shape example,minimum height=1.75cm] {};
  \foreach \anchor/\placement in {north/above, south/below,
                                  output/right, input/left}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

49.4.10 Implementation: The IEC-Style EE-Symbols Shape Library

TikZ Library shapes.gates.ee.IEC

```
\usepgflibrary{shapes.gates.ee.IEC}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.gates.ee.IEC]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.gates.ee.IEC} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.gates.ee.IEC] % ConTeXt when using TikZ
```

This library defines shapes for depicting ee symbols according to the IEC recommendations. These shapes will typically be used in conjunction with the graphic mechanism detailed earlier, but you can also used them directly.

Shape generic circle IEC

This shape inherits from `circle ee`, which in turn is just a normal `circle` with additional `input` and `output` anchors at the left and right ends. However, additionally, this shape allows you to specify a path that should be added before the background path using the following key:

/pgf/generic circle IEC/before background=⟨*code*⟩ (no default)

When a node of shape `generic circle IEC` is created, the current setting of this key is used as the "before background path". This means that after the circle's background has been drawn/filled/whatever, the ⟨*code*⟩ is executed.

When the ⟨*code*⟩ is executed, the coordinate system will have been transformed in such a way that the point (1pt, 0pt) lies at the right end of the circle and (0pt, 1pt) lies at the top of the circle. (More precisely, these points will lie exactly on the middle of the radial line.)

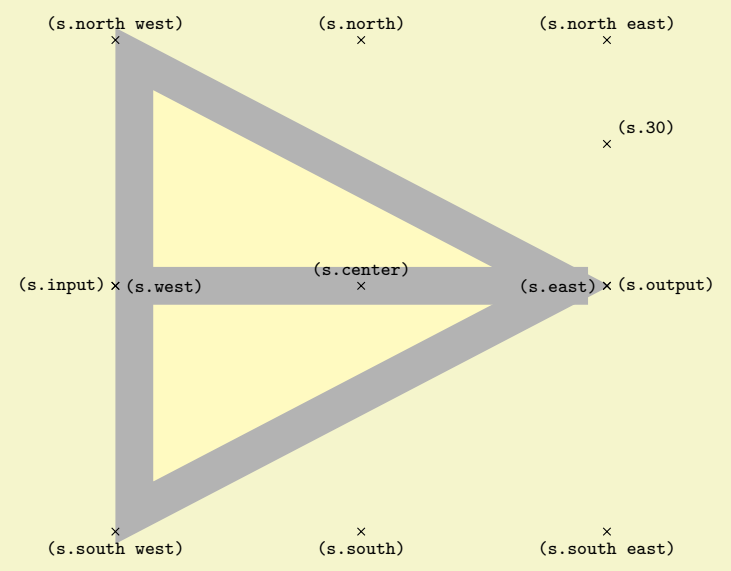
Here is an examples of how to use this shape:




```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node [generic circle IEC,
             /pgf/generic circle IEC/before background={
               \pgfpathmoveto{\pgfpointorigin}
               \pgfpathlineto{\pgfpoint{1pt}{0pt}}
               \pgfpathlineto{\pgfpoint{0pt}{1pt}}
               \pgfpathlineto{\pgfpoint{-0.5pt}{-0.5pt}}
               \pgfusepathqstroke
             },
             draw] {Hello world};
```

Shape generic diode IEC

This shape is used to depict diodes. The main shape is taken up by a "right pointing" triangle. The anchors are positioned on the border of a rectangle around the diode, see the below example. The diode's size is based on the current settings of `minimum width` and `minimum height`.



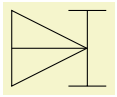

```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=generic diode IEC,shape example,minimum size=6cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

This shape, like the `generic circle IEC` shape, is generic in the sense that there is a special key that is used for the before background drawings:

`/pgf/generic diode IEC/before background=`⟨*code*⟩ (no default)

Similarly to the `generic circle IEC` shape, when a node of shape `generic diode IEC` is created, the current setting of this key is used as the "before background path". When the ⟨*code*⟩ is executed, the coordinate system will have been transformed in such a way that the origin is at the "tip" of the diode's triangle, the point (0pt, 1pt) is exactly half the diode's height above this origin, and the point (1pt, 0pt) is half the diode's height to the right of the origin.

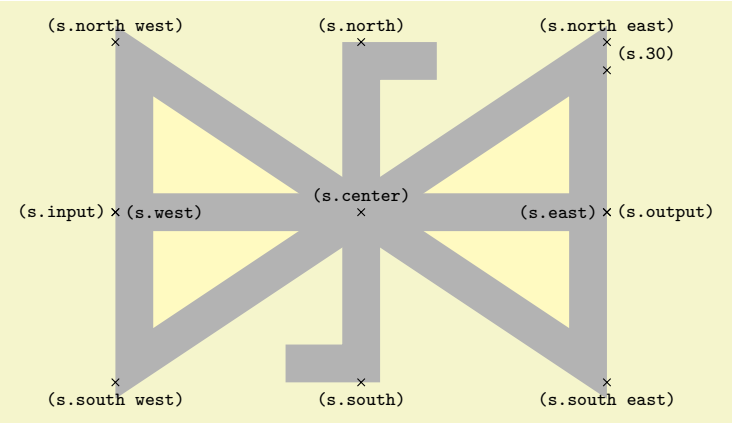
The idea is that you use this key to draw different kinds of diode endings.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node [minimum size=1cm,generic diode IEC,
             /pgf/generic diode IEC/before background={
               \pgfpathmoveto{\pgfqpoint{-.5pt}{-1pt}}
               \pgfpathlineto{\pgfqpoint{.5pt}{-1pt}}
               \pgfpathmoveto{\pgfqpoint{0pt}{-1pt}}
               \pgfpathlineto{\pgfqpoint{0pt}{1pt}}
               \pgfpathmoveto{\pgfqpoint{-.5pt}{1pt}}
               \pgfpathlineto{\pgfqpoint{.5pt}{1pt}}
               \pgfusepathqstroke
             },
             draw] {};
```

Shape `breakdown diode IEC`

This shape is used to depict a bidirectional breakdown diode. The diode's size is based on the current settings of `minimum width` and `minimum height`.

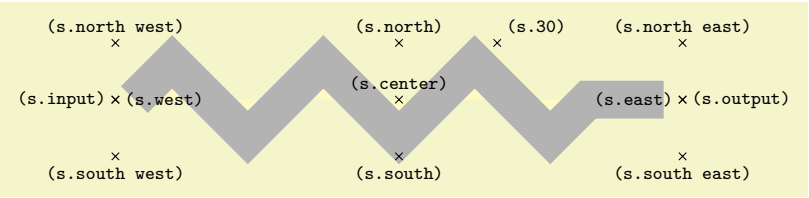



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=breakdown diode IEC,shape example,minimum width=6cm,minimum height=4cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `var resistor IEC`

This shape is used to depict a variant version of a resistor. Its size is computed as for a rectangle (thus, its size depends things like the `minimum height`). Then, inside this rectangle, a background path is set up according to the following rule: Starting from the left end, zigzag segments are added to the path. Each segment consists of a line at a 45 degree angle going up to the top of the rectangle, then going down to the bottom, then going up to mid height of the node. As many segments as possible are put inside as possible. The last segment is then connected to the output anchor via a straight line.

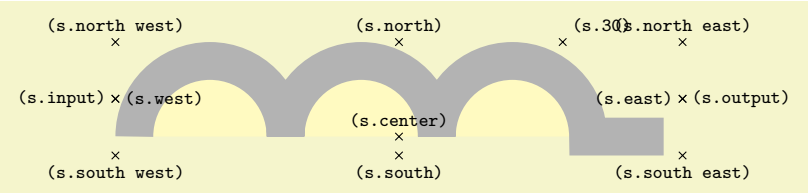
All of this means that, in general, the shape should be much wider than high.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=var resistor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `inductor IEC`

This shape is used to depict an inductor, using a bumpy line. Its size is computed as follows: Any text and `inner sep` are ignored (and should normally not be given). The `minimum height` plus (twice) the `outer ysep` specify the distance between the `north` and `south` anchors, similarly for the `minimum width` plus the `outer xsep` for the `east` and `west`. The bumpy line is drawn starting from the lower left corner to the lower right corner with bumps being half-circles whose height is exactly the `minimum height`. The `center` of the shape is just above the `south` anchor, at a distance of the `outer ysep`.

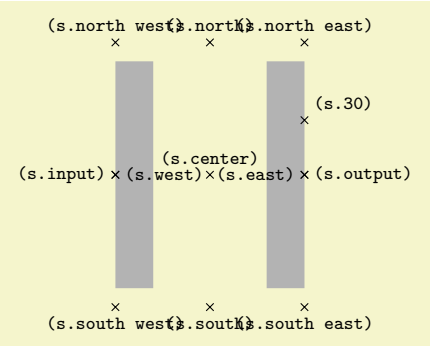



```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=inductor IEC,shape example,minimum width=7cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Just as for a `var resistor IEC`, as many bumps as possible are added and the last bump is connected to the output anchor via a straight line.

Shape capacitor IEC

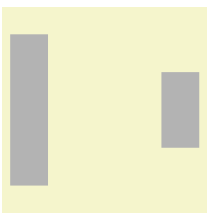
This shape is based on a `rectangle ee`. However, instead of a rectangle as the background path, only the "left and right lines" that make up the rectangle are drawn.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=capacitor IEC,shape example,
        minimum width=2cm,minimum height=3cm,inner sep=0pt] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape battery IEC

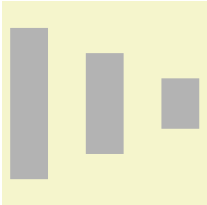
This shape is similar to a `capacitor IEC`, however, the right line is only half the height of the left line.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node[shape=battery IEC,shape example,minimum size=2cm,
            inner sep=0pt] {};
```

Shape ground IEC

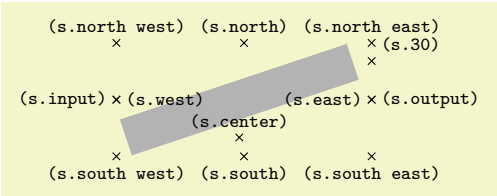
This shape is similar to a batter IEC, only three lines of different heights are drawn.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node[shape=ground IEC,shape example,minimum size=2cm,
            inner sep=0pt] {};
```

Shape make contact IEC

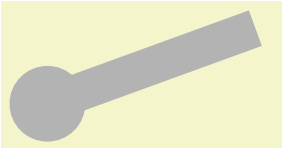
This shape consists of a line going from the lower left corner to the upper right corner. The size and anchors of this shape are computed in the same way as for an inductor IEC.




```
\usetikzlibrary {circuits.ee.IEC}
\begin{tikzpicture}
  \node[name=s,shape=make contact IEC,shape example,minimum width=3cm,minimum height=1cm] {};
  \foreach \anchor/\placement in
    {center/above, 30/above right,
     north/above, south/below, east/left, west/right,
     north east/above, south east/below, south west/below, north west/above,
     input/left,output/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape var make contact IEC

This shape works like make contact IEC, only a little circle is added to the path at the lower left corner. The radius of this circle is one twelfth of the width of the node.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node[shape=var make contact IEC,shape example,
            minimum height=1cm,minimum width=3cm,inner sep=0pt] {};
```

Shape break contact IEC

This shape depicts a contact that can be broken. It works like make contact IEC.



```
\usetikzlibrary {circuits.ee.IEC}
\tikz \node[shape=break contact IEC,shape example,
            minimum height=1cm,minimum width=3cm,inner sep=0pt] {};
```

50 Decoration Library

50.1 Overview and Common Options

The decoration libraries define a number of (more or less useful) decorations that can be applied to paths. The usage of decorations is not covered in the present section, please consult Sections 24, which explains how decorations are used in Ti*k*Z, and 103, which explains how new decorations can be defined.

The decorations are influenced by a number of parameters that can be set using the `decoration` option. These parameters are typically shared between different decorations. In the following, the general options are documented (they are defined directly in the `decoration` module), special-purpose keys are documented with the decoration that uses it.

Since you are encouraged to use these keys to make your own decorations configurable, it is indicated for each key where the value is stored (so that you can access it). Note that some values are stored in TEX dimension registers while others are stored in macros.

`/pgf/decoration/amplitude=`⟨*dimension*⟩ (no default, initially `2.5pt`)

This key determines the "desired height" (or amplitude) of decorations for which this makes sense. For instance, the initial value of `2.5pt` means that deforming decorations should deform a path by up to 2.5pt away from the original path.

This key sets the TEX-dimension `\pgfdecorationsegmentamplitude`.

`/pgf/decoration/meta-amplitude=`⟨*dimension*⟩ (no default, initially `2.5pt`)

This key determines the amplitude for a meta-decoration.

The key sets the TEX-macro (!) `\pgfmetadecorationsegmentamplitude`.

`/pgf/decoration/segment length=`⟨*dimension*⟩ (no default, initially `10pt`)

Many decorations are made up of small segments. This key determines the desired length of such segments.

This key sets the TEX-dimension `\pgfdecorationsegmentlength`.

`/pgf/decoration/meta-segment length=`⟨*dimension*⟩ (no default, initially `1cm`)

This determined the length of the meta-segments from which a meta-decoration is made up.

This key sets the TEX-macro (!) `\pgfmetadecorationsegmentlength`.

`/pgf/decoration/angle=`⟨*degree*⟩ (no default, initially `45`)

The way some decorations look like depends on a configurable angle. For instance, a `wave` decoration consists of arcs and the opening angle of these arcs is given by the `angle`.

This key sets the TEX-macro `\pgfdecorationsegmentangle`.

`/pgf/decoration/aspect=`⟨*factor*⟩ (no default, initially `0.5`)

For some decorations there is a natural aspect ratio. For instance, for a `brace` decoration the aspect ratio determines where the brace point will be.

This key sets the TEX-macro `\pgfdecorationsegmentaspect`.

`/pgf/decoration/start radius=`⟨*dimension*⟩ (no default, initially `2.5pt`)

For some decorations there is a natural start radius (of some circle, presumably).

This key stores the value directly inside the key.

`/pgf/decoration/end radius=`⟨*dimension*⟩ (no default, initially `2.5pt`)

For some decorations there is a natural end radius (of some circle, presumably).

This key stores the value directly inside the key.

`/pgf/decoration/radius=`⟨*dimension*⟩ (style, no default)

Sets the start and end radius simultaneously.

`/pgf/decoration/path has corners=`⟨*boolean*⟩ (no default, initially `false`)

This is a hint to the decoration code as to whether the path has corners or not. If a path has a sharp corner, setting this option to `true` may result in better rendering of the decoration because the joins of input segments are approached "more carefully" than when this key is set to false. However, if the path is, say, a smooth circle, setting this key to `true` will usually look worse. Most decorations ignore this key, anyway. Internally, it sets the TEX-if `\ifpgfdecoratepathhascorners`.

50.2 Handling "Dimension too large" errors

In case you should run into a "Dimension too large error" when using the `decorations` libraries, there is a pretty high chance that you can resolve this by using the `fpu` library in combination with `/pgf/fpu/install only={reciprocal}`. Please note that this key should only be applied locally to avoid other errors (see also at the definition of this key on page 690).



```
\usetikzlibrary {decorations.markings,fpu}
\begin{tikzpicture}
  \begin{scope}[/pgf/fpu/install only={reciprocal}]
    \draw[postaction=decorate,decoration={
        markings,mark=at position 0.52 with {
            \draw circle[radius=2pt];
        }},
    ] plot[smooth,variable=\x,domain=-1:1] (\x*\x*\x,\x*\x);
  \end{scope}
\end{tikzpicture}
```

50.3 Path Morphing Decorations

TikZ Library `decorations.pathmorphing`

```
\usepgflibrary{decorations.pathmorphing}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.pathmorphing]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.pathmorphing} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.pathmorphing] % ConTeXt when using TikZ
```

A *path morphing decoration* "morphs" or "deforms" the to-be-decorated path. This means that what used to be a straight line might afterwards be a snaking curve and have bumps. However, a line is still a line and path deforming decorations do not change the number of subpaths. For instance, if the path used to consist of two circles and an open arc, the path will, after the decoration process, still consist of two closed subpaths and one open subpath.

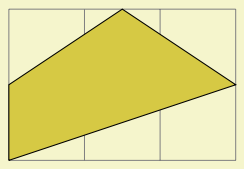
50.3.1 Decorations Producing Straight Line Paths

The following deformations use only straight lines in order to morph the paths.

Decoration `lineto`

This decoration replaces the path by straight lines. For each curve, the path simply goes directly from the start point to the end point. In the following example, the arc actually consist of two subcurves.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.



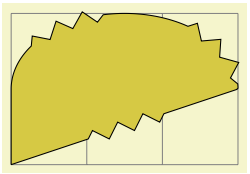
```
\usetikzlibrary {decorations}
\begin{tikzpicture}[decoration=lineto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `straight zigzag`

This (meta-)decoration decorates the path by alternating between `curveto` and `zigzag` decorations. It always finishes with the `curveto` decoration. The following parameters influence the decoration:

- `amplitude` determines how much the zigzag line raises above and falls below a straight line to the target point.

- `segment length` determines the length of a complete “up-down” cycle.
- `meta-segment length` determines the length of the `curveto` and the `zigzag` decorations.

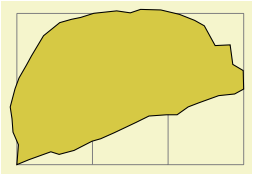


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration={straight zigzag,meta-segment length=1.1cm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `random steps`

This decoration consists of straight line segments. The line segments head towards the target, but each step is randomly shifted a little bit. The following parameters influence the decorations:

- `segment length` determines the basic length of each step.
- `amplitude` The end of each step is perturbed both in x- and in y-direction by two values drawn uniformly from the interval $[-d, d]$, where d is the value of `amplitude`.

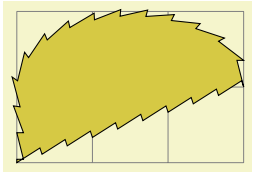


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
    [decoration={random steps,segment length=2mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `saw`

This decoration looks like the blade of a saw. The following parameters influence the decoration:

- `amplitude` determines how much each spike raises above the straight line.
- `segment length` determines the length each spike.

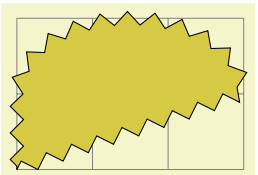


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=saw]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration `zigzag`

This decoration looks like a zigzag line. The following parameters influence the decoration:

- `amplitude` determines how much the zigzag line raises above and falls below a straight line to the target point.
- `segment length` determines the length of a complete “up-down” cycle.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=zigzag]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

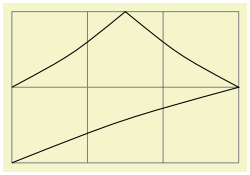
50.3.2 Decorations Producing Curved Line Paths

Decoration bent

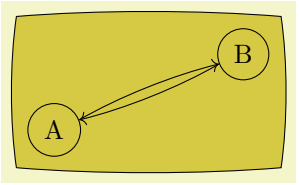
This decoration adds a slightly bent line from the start to the target. The amplitude of the bend is given `amplitude` (an amplitude of zero gives a straight line).

- `amplitude` determines the amplitude of the bend.
- `aspect` determines how tight the bend is. A good value is around `0.3`.

Note that this decoration makes only little sense for curves. You should apply it only to straight lines.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=bent]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) -- (1.5,2) -- (0,1);
\end{tikzpicture}
```

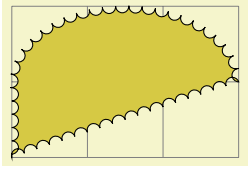



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration={bent,aspect=.3}]
  \draw [decorate,fill=yellow!80!black] (0,0) rectangle (3.5,2);
  \node[circle,draw] (A) at (.5,.5) {A};
  \node[circle,draw] (B) at (3,1.5) {B};
  \draw[->,decorate] (A) -- (B);
  \draw[->,decorate] (B) -- (A);
\end{tikzpicture}
```

Decoration bumps

This decoration replaces the path by little half ellipses. The following parameters influence it.

- `amplitude` determines the height of the half ellipse.
- `segment length` determines the width of the half ellipse.

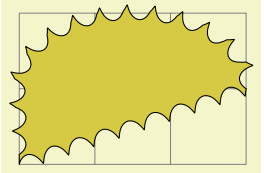


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=bumps]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

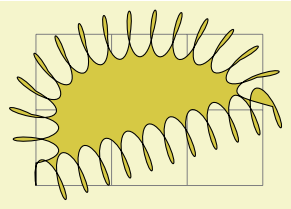
Decoration coil

This decoration replaces the path by a coiled line. To understand how this works, imagine a three-dimensional spring. The spring's axis points along the path toward the target. Then, we "view" the spring from a certain angle. If we look "straight from the side" we will see a perfect sine curve, if we look "more from the front" we will see a coil. The following parameters influence the decoration:

- `amplitude` determines how much the coil rises above the path and falls below it. Thus, this is the radius of the coil.
- `segment length` determines the distance between two consecutive "curls". Thus, when the spring is see "from the side" this will be the wave length of the sine curve.
- `aspect` determines the "viewing direction". A value of `0` means "looking from the side" and a value of `0.5`, which is the default, means "look more from the front".



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=coil]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

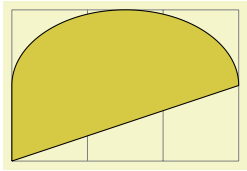


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}
    [decoration={coil,aspect=0.3,segment length=3mm,amplitude=3mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration curveto

This decoration simply yields a line following the original path. This means that (ideally) it does not change the path and follows any curves in the path (hence the name). In reality, due to the internals of how decorations are implemented, this decoration actually replaces the path by numerous small straight lines.

This decoration is mostly useful in conjunction with meta-decorations. It is also actually defined in the decoration module and is always available.

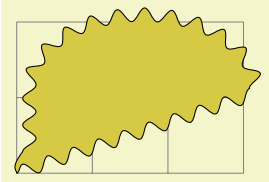


```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=curveto]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

Decoration snake

This decoration replaces the path by a line that looks like a snake seen from above. More precisely, the snake is a sine wave with a "softened" start and ending. The following parameters influence the snake:

- `amplitude` determines the sine wave's amplitude.
- `segment length` determines the sine wave's wavelength.



```
\usetikzlibrary {decorations.pathmorphing}
\begin{tikzpicture}[decoration=snake]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black]
    (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

50.4 Path Replacing Decorations

TikZ Library decorations.pathreplacing

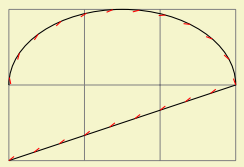
```
\usepgflibrary{decorations.pathreplacing}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.pathreplacing]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.pathreplacing} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.pathreplacing] % ConTeXt when using TikZ
```

This library defines decorations that replace the to-be-decorated path by another path. Unlike morphing decorations, the replaced path might be quite different, for instance a straight line might be replaced by a set of circles. Note that filling a path that has been replaced using one of the decorations in this library typically does not fill the original area but, rather, the smaller area of the newly-created path segments.

Decoration border

This decoration adds straight lines to the path that are at a specific angle to the line toward the target. The idea is to add these little lines to indicate the "border" of an area. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive ticks.
- `amplitude` determines the length of the ticks.
- `angle` determines the angle between the ticks and the line of the path.

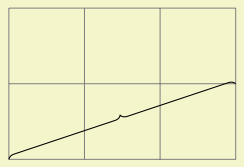


```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration=border]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate,draw,red}]
        (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `brace`

This decoration replaces a straight line path by a long brace. The left and right end of the brace will be exactly on the start and endpoint of the decoration. The decoration really only makes sense for paths that are a straight line.

- `amplitude` determines how much the brace rises above the path.
- `aspect` determines the fraction of the total length where the "middle part" of the brace will be.

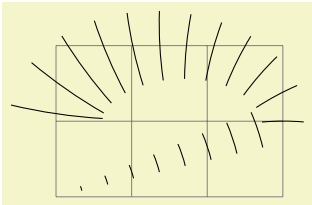


```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration=brace]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1);
\end{tikzpicture}
```

Decoration `expanding waves`

This decoration adds arcs to the path that get bigger along the line towards the target. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive arcs.
- `angle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration={expanding waves,angle=5}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `moveto`

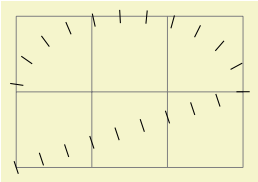
This decoration simply jumps to the end of the path using a move-to path operation. It is mainly useful as `pre=moveto` or `post=moveto` decorations.

This decoration is actually always defined when the decoration module is loaded, but it is documented here for consistency.

Decoration `ticks`

This decoration replaces the path by straight lines that are orthogonal to the path. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive ticks.
- `amplitude` determines half the length of the ticks.

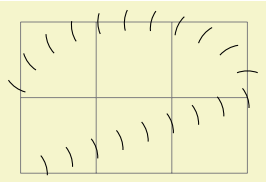


```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration=ticks]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `waves`

This decoration replaces the path by arcs that have a constant size. The following parameters influence the decoration:

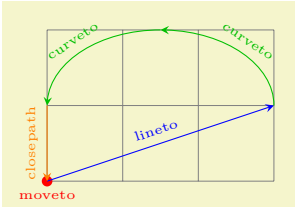
- `segment length` determines the distance between consecutive arcs.
- `angle` determines the opening angle below and above the path. Thus, the total opening angle is twice this angle.
- `radius` determines the radius of each arc.



```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[decoration={waves,radius=4mm}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration `show path construction`

This decoration allows "something different" to be done for each *type* of input segment (i.e., moveto, lineto, curveto or closepath). Typically, each segment will be replaced with another path, but this need not necessarily be the case.




```
\usetikzlibrary {decorations.pathreplacing}
\begin{tikzpicture}[>=stealth, every node/.style={midway, sloped, font=\tiny},
  decoration={show path construction,
    moveto code={
      \fill [red] (\tikzinputsegmentfirst) circle [radius=2pt]
        node [fill=none, below] {moveto};},
    lineto code={
      \draw [blue,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
        node [above] {lineto};
    },
    curveto code={
      \draw [green!75!black,->] (\tikzinputsegmentfirst) .. controls
        (\tikzinputsegmentsupporta) and (\tikzinputsegmentsupportb)
        ..(\tikzinputsegmentlast) node [above] {curveto};
    },
    closepath code={
      \draw [orange,->] (\tikzinputsegmentfirst) -- (\tikzinputsegmentlast)
        node [above] {closepath};}
  }]
  \draw [help lines] grid (3,2);
  \path [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1) -- cycle;
\end{tikzpicture}
```

The following keys can be used to specify the code to execute for each type of input segment.

`/pgf/decoration/moveto code=`⟨*code*⟩ (no default, initially `{}`)

Set the code to be executed for every moveto input segment. It is important to remember that the transformations applied by the decoration automaton are turned *off* when ⟨*code*⟩ is executed.

`/pgf/decoration/lineto code=`⟨*code*⟩ (no default, initially `{}`)

Set the code to be executed for every lineto input segment.

/pgf/decoration/curveto code=⟨*code*⟩ (no default, initially {})

Set the code to be executed for every curveto input segment.

/pgf/decoration/closepath code=⟨*code*⟩ (no default, initially {})

Set the code to be executed for every closepath input segment.

Within ⟨*code*⟩ the first and last points on the current input segment can be accessed using `\pgfpointdecoratedinputsegmentfirst` and `\pgfpointdecoratedinputsegmentlast`. For curves, the control (support) points can be accessed using `\pgfpointdecoratedinputsegmentsupporta` and `\pgfpointdecoratedinputsegmentsupportb`.

In Ti*k*Z, you can use the following macros inside a Ti*k*Z coordinate.

`\tikzinputsegmentfirst`

The first point on the current input segment path.

`\tikzinputsegmentlast`

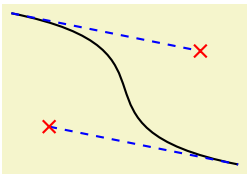
The last point on the current input segment path.

`\tikzinputsegmentsupporta`

The first support on the curveto input segment path.

`\tikzinputsegmentsupportb`

The second support on the curveto input segment path.



```
\usetikzlibrary {decorations.pathreplacing,shapes.misc}
\tikzset{
  show curve controls/.style={
    decoration={
      show path construction,
      curveto code={
        \draw [blue, dashed]
          (\tikzinputsegmentfirst)    -- (\tikzinputsegmentsupporta)
          node [at end, cross out, draw, solid, red, inner sep=2pt]{};
        \draw [blue, dashed]
          (\tikzinputsegmentsupportb) -- (\tikzinputsegmentlast)
          node [at start, cross out, draw, solid, red, inner sep=2pt]{};
      }
    },decorate
  }
}

\begin{tikzpicture}
  \draw [postaction=show curve controls, thick]
    (0,2) .. controls (2.5,1.5) and (0.5,0.5) .. (3,0);
\end{tikzpicture}
```

50.5 Marking Decorations

50.5.1 Overview

A *marking on a path* is any kind of graphic that is placed on a specific position on a path. Markings are useful in rather diverse situations: you can use them to, say, place little "footsteps" along a path as if someone where walking along the path; to place arrow tips on the middle of a path to indicate the "direction" in which something is flowing; or you can use them to place informative information at certain positions of a path.

For historical reasons there are three different libraries for placing marks on a path. They differ in what kind of markings can be added to a path. We start with the most general and most useful of these libraries.

50.6 Arbitrary Markings

Ti*k*Z Library `decorations.markings`

```
\usepgflibrary{decorations.markings}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.markings]  % ConTeXt when using pure PGF
```

```
\usetikzlibrary{decorations.markings} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.markings] % ConTeXt when using TikZ
```

Markings are arbitrary "marks" that can be put on a path. Marks can be arrow tips or nodes or even whole pictures.

Decoration markings

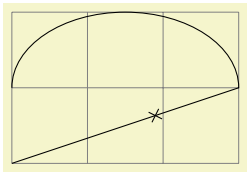
A *marking* can be thought of a "little picture" or more precisely of "some scope contents" that is placed "on" a path at a certain position. Suppose the marking should be a simple cross. We can produce this with the following code:

```
\draw (-2pt,-2pt) -- (2pt,2pt);
\draw (2pt,-2pt) -- (-2pt,2pt);
```

If we use this code as a marking at position `2cm` on a path, then the following happens: PGF determines the position on the path that is 2cm along the path. Then it translates the coordinate system to this position and rotates it such that the positive x-axis is tangent to the path. Then a protective scope is created, inside which the above code is executed – resulting in a little cross on the path.

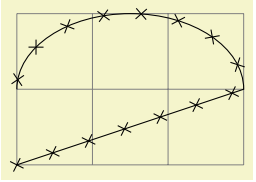
The `markings` decoration allows you to place one or more such markings on a path. The decoration destroys the input path (except in certain cases, detailed later), which means that it uses the path for determining positions on the path, but after the decoration is done this path is gone. You typically need to use a `postaction` to add markings.

Let us start with the above example in real code:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=% actually add a mark
      at position 2cm
      with
      {
        \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);
      }
    }
    ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

We can also add the cross repeatedly:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=% actually add a mark
      between positions 0 and 1 step 5mm
      with
      {
        \draw (-2pt,-2pt) -- (2pt,2pt);
        \draw (2pt,-2pt) -- (-2pt,2pt);
      }
    }
    ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

The `mark` decoration option is used to specify a marking. It comes in two versions:

`/pgf/decoration/mark=at position` ⟨*pos*⟩ `with` ⟨*code*⟩ (no default)

The options specifies that when a `marking` decoration is applied, there should be a marking at position ⟨*pos*⟩ on the path whose code is given by ⟨*code*⟩.

The ⟨*pos*⟩ can have four different forms:

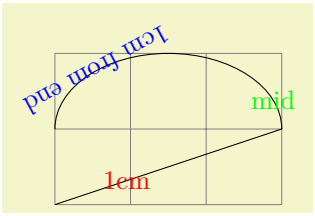
1. It can be a non-negative dimension like `0pt` or `2cm` or `5cm/2`. In this case, it refers to the position along the path that is this far displaced from the start.

2. It can be a negative dimension like `-1cm-2pt` or `-1sp`. In this case, the position is taken from the end of the path. Thus, `-1cm` is the position that is −1cm displaced from the end of the path.
3. It can be a dimensionless non-negative number like `1/2` or `0.333+2*0.1`. In this case, the ⟨*pos*⟩ is interpreted as a factor of the total path length. Thus, a ⟨*pos*⟩ or `0.5` refers to the middle of the path, `0.1` is near the start, and so on.
4. It can be a dimensionless negative number like `-0.1`. Then, again, the fraction of the path length counts "from the end".

The ⟨*pos*⟩ determines a position on the path. When the marking is applied, the (high level) coordinate system will have been transformed so that the origin lies at this position and the positive x-axis points along the path. For this coordinate system, the ⟨*code*⟩ is executed. It can contain all sorts of graphic drawing commands, including (even named) nodes.

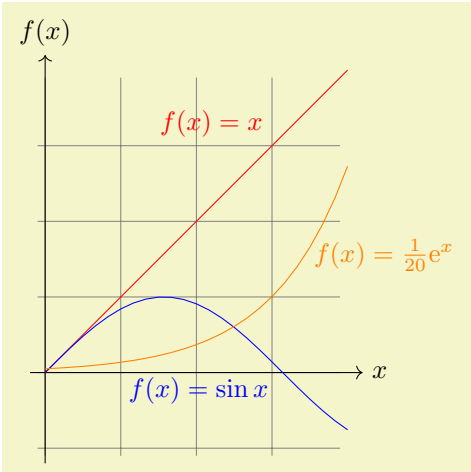
If the position lies past the end of the path (for instance if ⟨*pos*⟩ is set to `1.2`), the marking will not be drawn.

It is possible to give the `mark` option several times, which causes several markings to be applied. In this case, however, it is necessary that the positions on the path are in increasing order. That is, it is not allowed (and will result in chaos) to have a marking that lies earlier on the path to follow a marking that is later on the path.




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=at position 1cm  with \node[red]{1cm};,
    mark=at position .5   with \node[green]{mid};,
    mark=at position -1cm with {\node[blue,transform shape]{1cm from end};}}
    ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Here is an example that shows how markings can be used to place text on plots:




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[domain=0:4,label/.style={postaction={
      decorate,
      decoration={
        markings,
        mark=at position .75 with \node #1;}}}]
  \draw[very thin,color=gray] (-0.1,-1.1) grid (3.9,3.9);

  \draw[->] (-0.2,0) -- (4.2,0) node[right] {$x$};
  \draw[->] (0,-1.2) -- (0,4.2) node[above] {$f(x)$};

  \draw[red,label={[above left]{$f(x)=x$}}]                        plot (\x,\x);
  \draw[blue,label={[below left]{$f(x)=\sin x$}}]                   plot (\x,{sin(\x r)});
  \draw[orange,label={[right]{$f(x)= \frac{1}{20} \mathrm e^x$}}] plot (\x,{0.05*exp(\x)});
\end{tikzpicture}
```

When the ⟨*code*⟩ is being executed, two special keys will have been set up, whose value may be of interest:

`/pgf/decoration/mark info/sequence number` (no value)

This key can only be read. Its value (which can be obtained using the `\pgfkeysvalueof` command) is a "sequence number" of the mark. The first mark that is added to a path has number `1`, the second number `2`, and so on. This key is mainly useful in conjunction with repeated markings (see below).

`/pgf/decoration/mark info/distance from start` (no value)

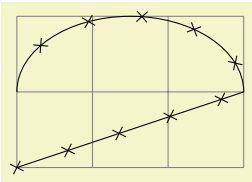
This key can only be read. Its value is the distance of the marking from the start of the path in points. For instance, if the path length is 100pt and the marking is in the middle of the path, the value of this key would be `50.0pt`.

A second way to use the `mark` key is the following:

`/pgf/decoration/mark=between positions` ⟨*start pos*⟩ `and` ⟨*end pos*⟩ `step` ⟨*stepping*⟩ `with` ⟨*code*⟩ (no default)

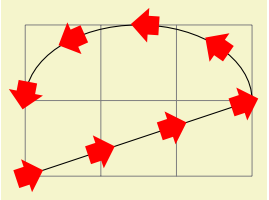
This works similarly to the `at position` version of this option, only multiple marks are placed, starting at ⟨*start pos*⟩ and then spaced apart by ⟨*stepping*⟩. The ⟨*start pos*⟩, the ⟨*end pos*⟩, and also the ⟨*stepping*⟩ may all be specified in the same way as for the `at position` version, that is, either using units or no units and also using positive or negative values.

Let us start with a simple example in which we place ten crosses along a path starting with the beginning of the path (⟨*start pos*⟩ = 0) and ending at the end (⟨*end pos*⟩ = 1).



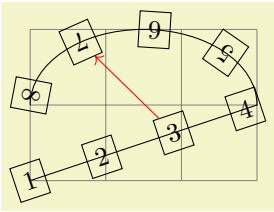
```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 0.1
         with { \draw (-2pt,-2pt) -- (2pt,2pt);
                \draw (2pt,-2pt) -- (-2pt,2pt); } } ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

In the next example we place arrow shapes on the path instead of crosses. Note the use of the `transform shape` option to ensure that the nodes are actually rotated.



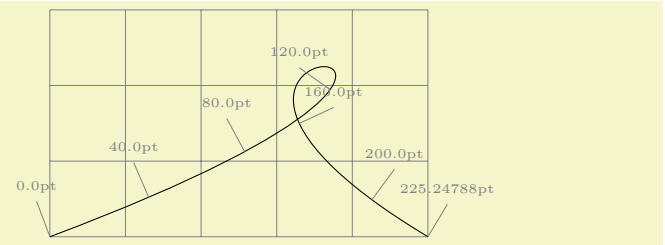
```
\usetikzlibrary {decorations.markings,shapes.arrows}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 1cm
      with { \node [single arrow,fill=red,
                    single arrow head extend=3pt,transform shape] {};}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Using the key `sequence number` we can also "number" the nodes and even refer to them later on.




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark=between positions 0 and 1 step 1cm with {
      \node [draw,
        name=mark-\pgfkeysvalueof{/pgf/decoration/mark info/sequence number},
        transform shape]
      {\pgfkeysvalueof{/pgf/decoration/mark info/sequence number}};}}]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
  \draw [red,->] (mark-3) -- (mark-7);
\end{tikzpicture}
```

In the following example we use the distance info to place "length information" on a path:




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    % Main marks
    mark=between positions 0 and 1 step 40pt with
      { \draw [help lines] (0,0) -- (0,0.5)
        node[above,font=\tiny]{
          \pgfkeysvalueof{/pgf/decoration/mark info/distance from start}}; },
    mark=at position -0.1pt with
      { \draw [help lines] (0,0) -- (0,0.5)
        node[above,font=\tiny]{
          \pgfkeysvalueof{/pgf/decoration/mark info/distance from start}}; }}]
  \draw [help lines] grid (5,3);
  \draw [postaction={decorate}]  (0,0) .. controls (8,3) and (0,3) .. (5,0) ;
\end{tikzpicture}
```

`/pgf/decoration/reset marks` (no value)

Since `mark` options accumulate, there needs to be a way to "reset" things, so that any `mark` options set in an enclosing scope do not interfere. This option does exactly this. Note that when the ⟨*code*⟩ of a marking is executed, the markings are automatically reset.

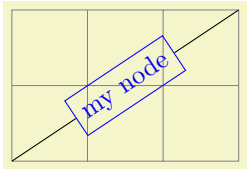
As mentioned earlier, the decoration usually destroys the path. However, this is no longer the case when the following key is set:

`/pgf/decoration/mark connection node=`⟨*node name*⟩ (no default, initially `empty`)

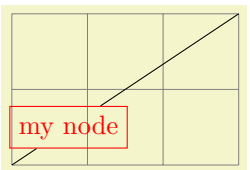
When this key is set to a non-empty ⟨*node name*⟩ while the decoration is being processed, the following happens: The marking code should, among possibly other things, define a node named ⟨*node name*⟩. Then, the output path of this decoration will contain a line-to to "one end" of this node, followed by a moveto to the "other end" of the node. More precisely, the first end is given by the position on the border of ⟨*node name*⟩ that lies in the direction "from which the path heads toward the node" while the other end lies on the border "where the path heads away from the node". Furthermore, this option causes the decoration to end with a line-to to the end instead of a move-to.

The net effect of all this is that when you decorate a straight line with one or more markings that contain just a node, the line will effectively connect these nodes.

Here are two examples that show how this works:




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .5 with
      {\node [draw,blue,transform shape] (my node) {my node};}}]
  \draw [help lines] grid (3,2);
  \draw decorate { (0,0) -- (3,2) };
\end{tikzpicture}
```




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={markings,
    mark connection node=my node,
    mark=at position .25 with
      {\node [draw,red] (my node) {my node};}}]
  \draw [help lines] grid (3,2);
  \draw decorate { (0,0) -- (3,2) };
\end{tikzpicture}
```

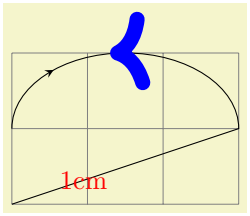
50.6.1 Arrow Tip Markings

Frequent markings that are hard to create correctly are arrow tips. For them, two special commands are available when the ⟨*code*⟩ of a `mark` option is executed. (They are only defined in this code):

`\arrow[`⟨*options*⟩`]{`⟨*arrow end tip*⟩`}`

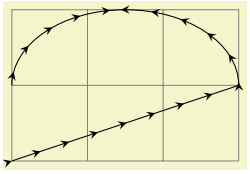
This command simply draws the ⟨*arrow end tip*⟩ at the origin, pointing right. This is exactly what you need when you want to draw an arrow tip as a marking.

The ⟨*options*⟩ can only be given when Ti*k*Z is used. In this case, they are executed in a scope that contains the arrow tip.




```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=at position 1cm  with {\node[red]{1cm};},
    mark=at position .75  with {\arrow[blue,line width=2mm]{>}},
    mark=at position -1cm with {\arrowreversed[black]{stealth}}}
    ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Here is a more useful example:



```
\usetikzlibrary {decorations.markings}
\begin{tikzpicture}[decoration={
    markings,% switch on markings
    mark=between positions 0 and .75 step 4mm with {\arrow{stealth}},
    mark=between positions .75 and 1 step 4mm with {\arrowreversed{stealth}}}
    ]
  \draw [help lines] grid (3,2);
  \draw [postaction={decorate}] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

`\arrowreversed[`⟨*options*⟩`]{`⟨*arrow end tip*⟩`}`

As above, only the arrow end tip is flipped and points in the other direction.

50.6.2 Footprint Markings

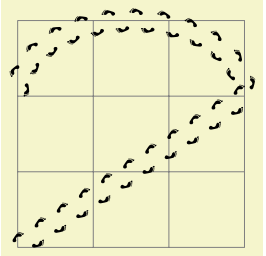
Ti*k*Z Library `decorations.footprints`

```
\usepgflibrary{decorations.footprints}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.footprints]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.footprints} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.footprints] % ConTeXt when using TikZ
```

The decorations of this library can be used to decorate a path with little footprints, as if someone had "walked" along the path.

Decoration `footprints`

The footprint decoration adds little footprints around the path. They start with the left foot.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot length=5pt,stride length=10pt}]
  \draw [help lines] grid (3,3);
  \fill [decorate] (0,0) -- (3,2) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

You can influence the way this decoration looks using the following options:

/pgf/decoration/foot length (initially 10pt)

The length or size of the footprint itself. A larger value makes the footprint larger, but does not change the stride length.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot length=20pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/stride length (initially 30pt)

The length of strides. This is the distance between the beginnings of left footprints along the path.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,stride length=50pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot sep (initially 4pt)

The separation in the middle between the footprints. The footprints are moved away from the path by half this amount.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot sep=10pt}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot angle (initially 10)

Footprints are rotated by this much.



```
\usetikzlibrary {decorations.footprints}
\begin{tikzpicture}[decoration={footprints,foot angle=60}]
  \fill [decorate] (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/foot of (initially human)

The species whose footprints are shown. Possible values are:

Species	*Result*
gnome	
human	
bird	
felis silvestris	

50.6.3 Shape Background Markings

The third library for adding markings uses the background paths of certain shapes. This library is included mostly for historical reasons, using the `markings` library is usually preferable.

TikZ Library decorations.shapes

```
\usepgflibrary{decorations.shapes}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.shapes]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.shapes} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.shapes] % ConTeXt when using TikZ
```

This library defines decorations that use shapes or shape-like drawings to decorate a path. The following options are common options used by the decorations in this library:

`/pgf/decoration/shape width`=⟨*dimension*⟩ (no default, initially `2.5pt`)

The desired width of the shapes. For decorations that support varying shape sizes, this key sets both the start and end width (which can be overwritten using options like `shape start width`).

`/pgf/decoration/shape height`=⟨*dimension*⟩ (no default, initially `2.5pt`)

Works like the previous key, only for the height.

`/pgf/decoration/shape size`=⟨*dimension*⟩ (no default)

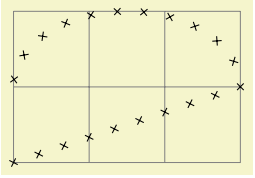
Sets the desired width and height simultaneously.

For the exact places and macros where these keys store the values, please consult the beginning of the code of the library.

Decoration crosses

This decoration replaces the path by (diagonal) crosses. The following parameters influence the decoration:

- `segment length` determines the distance between (the centers of) consecutive crosses.
- `shape height` determines the height of each cross.
- `shape width` determines the width of each cross.

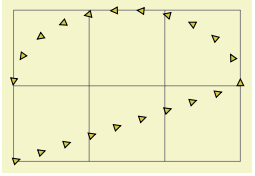


```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}[decoration=crosses]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

Decoration triangles

This decoration replaces the path by triangles that point along the path. The following parameters influence the decoration:

- `segment length` determines the distance between consecutive triangles.
- `shape height` determines the height of the triangle side that is orthogonal to the path.
- `shape width` determines the width of the triangle.

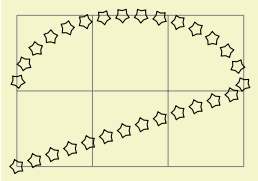


```
\usetikzlibrary {decorations.shapes}
\begin{tikzpicture}[decoration=triangles]
  \draw [help lines] grid (3,2);
  \draw [decorate,fill=yellow!80!black] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

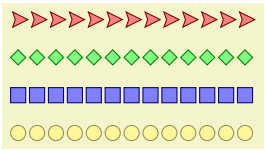
Decoration shape backgrounds

This is a general decoration that replaces the to-be-decorated path by repeated copies of the background path of an arbitrary shape that has previously been defined using the `\pgfdeclareshape` command (that is, you can use any shape in the shape libraries).

Please note that the background path of the shapes is used, but *no nodes are created*. This means that *you cannot have text inside the shapes of this path, you cannot name them, or refer to them.* Finally, this decoration *will not work with shapes that depend strongly on the size of the text box (like the arrow shapes).* If any of these restrictions pose a problem, use the `markings` library instead.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\begin{tikzpicture}[decoration={shape backgrounds,shape=star,shape size=5pt}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```



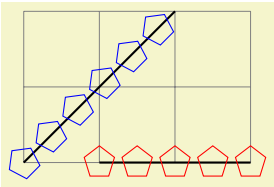
```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\tikzset{paint/.style={ draw=#1!50!black, fill=#1!50 },
         decorate with/.style=
           {decorate,decoration={shape backgrounds,shape=#1,shape size=2mm}}}
\begin{tikzpicture}
  \draw [decorate with=dart,      paint=red]    (0,1.5) -- (3,1.5);
  \draw [decorate with=diamond,   paint=green]  (0,1)   -- (3,1);
  \draw [decorate with=rectangle, paint=blue]   (0,0.5) -- (3,0.5);
  \draw [decorate with=circle,    paint=yellow] (0,0)   -- (3,0);
\end{tikzpicture}
```

All shapes are positioned by the anchor that is specified via the `anchor` decoration option:

`/pgf/decoration/anchor`=⟨*anchor*⟩ (no default, initially `center`)

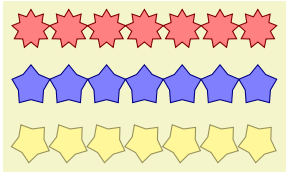
The anchor used to position the shape backgrounds.

A shape background path is added at the start point of the path and, if the distance between the shapes is appropriate, at the end point of the path.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\begin{tikzpicture}[decoration={
      shape backgrounds,shape=regular polygon,shape size=4mm}]
  \draw [help lines] grid (3,2);
  \draw [thick] (0,0) -- (2,2) (1,0) -- (3,0);
  \draw [red, decorate, decoration={shape sep=.5cm}]  (1,0) -- (3,0);
  \draw [blue, decorate, decoration={shape sep=.5cm}] (0,0) -- (2,2);
\end{tikzpicture}
```

Keys for customizing specific shapes can be specified (e.g., `star points`, `cloud puffs`, `kite angles`, and so on). The size of the shape is "enforced" using transformations. This means that the shape is typeset with an empty text box and some default size values, resulting in an initial shape. This shape is then rescaled using coordinate transformations so that it has the desired size (which may vary as we travel along the to-be-decorated path). This means that settings involving angles and distances may not appear entirely accurate. More general options such as `inner sep` and `minimum size` will be ignored, but transformations can be applied to each segment as described below.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50},
  my star/.style={decorate,decoration={shape backgrounds,shape=star},
                  star points=#1}
}
\begin{tikzpicture}[decoration={shape sep=.5cm, shape size=.5cm}]
  \draw [my star=9, paint=red]                              (0,1.5) -- (3,1.5);
  \draw [my star=5, paint=blue]                             (0,.75) -- (3,.75);
  \draw [my star=5, paint=yellow, shape border rotate=30] (0,0) -- (3,0);
\end{tikzpicture}
```

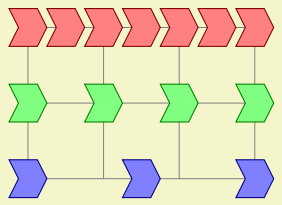
There are various keys to control the drawing of the shape decoration.

`/pgf/decoration/shape`=⟨*shape name*⟩ (no default, initially `circle`)

The shape whose background path is used.

/pgf/decoration/shape sep=⟨*spacing*⟩ (no default, initially `.25cm, between centers`)

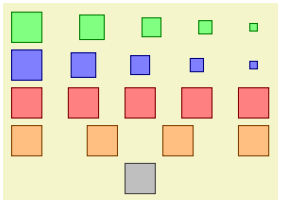
Set the spacing between the shapes on the decorations path. This can be just a distance on its own, but the additional keywords `between centers`, and `between borders` (which must be preceded by a comma), specify that the distance is between the center anchors of the shapes or between the edges of the *boundaries* of the shape borders.



```
\usetikzlibrary {decorations.shapes,shapes.symbols}
\begin{tikzpicture}[
    decoration={shape backgrounds,shape size=.5cm,shape=signal},
    signal from=west, signal to=east,
    paint/.style={decorate, draw=#1!50!black, fill=#1!50}]
  \draw [help lines] grid (3,2);
  \draw [paint=red, decoration={shape sep=.5cm}]
    (0,2) -- (3,2);
  \draw [paint=green, decoration={shape sep={1cm, between centers}}]
    (0,1) -- (3,1);
  \draw [paint=blue, decoration={shape sep={1cm, between borders}}]
    (0,0) -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/shape evenly spread=⟨*number*⟩ (no default)

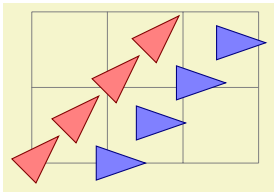
This key overrides the `shape sep` key and forces the decoration to fit ⟨*number*⟩ shapes evenly across the path. If ⟨*number*⟩ is less than `1`, then no shapes will be used. If ⟨*number*⟩ equals `1`, then one shape is put in the middle of the path. The additional keywords `by centers` (the default, if no keyword is specified) and `by borders` can be used (both preceded by a comma), to specify how the distance between shapes is determined. These keywords will only have a noticeable effect if the shapes sizes differ over time.



```
\usetikzlibrary {decorations.shapes}
\tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50},
  spreading/.style={
    decorate,decoration={shape backgrounds, shape=rectangle,
    shape start size=4mm,shape end size=1mm,shape evenly spread={#1}}}
}
\begin{tikzpicture}
  \fill [paint=green,spreading={5, by borders},
         decoration={shape scaled}]            (0,2)   -- (3,2);
  \fill [paint=blue,spreading={5, by centers},
         decoration={shape scaled}]            (0,1.5) -- (3,1.5);
  \fill [paint=red,    spreading=5]            (0,1)   -- (3,1);
  \fill [paint=orange, spreading=4]            (0,.5)  -- (3,.5);
  \fill [paint=gray,    spreading=1]            (0,0)   -- (3,0);
\end{tikzpicture}
```

/pgf/decoration/shape sloped=⟨*boolean*⟩ (no default, initially `true`)

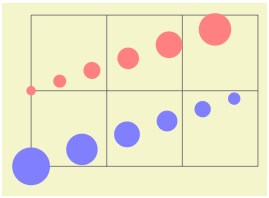
By default, shapes are rotated to the slope of the decorations path. If ⟨*boolean*⟩ is the value `false`, then this rotation is turned off. Internally this sets the TEX-if `\ifpgfshapedecorationsloped` accordingly.



```
\usetikzlibrary {decorations.shapes,shapes.geometric}
\tikzset{
  paint/.style={draw=#1!50!black, fill=#1!50}
}
\begin{tikzpicture}[decoration={
    shape width=.65cm, shape height=.45cm,
    shape=isosceles triangle, shape sep=.75cm,
    shape backgrounds}]
  \draw [help lines] grid (3,2);
  \draw [paint=red,decorate] (0,0) -- (2,2);
  \draw [paint=blue,decorate,decoration={shape sloped=false}]
                              (1,0) -- (3,2);
\end{tikzpicture}
```

It is possible to scale the width and height of the shapes along the length of the decorations path. The shapes are scaled between the starting size and the ending size. The following keys customize the way the decoration shapes are scaled:

/pgf/decoration/shape scaled=⟨*boolean*⟩ (no default, initially `false`)



```
\usetikzlibrary {decorations.shapes}
\tikzset{
  bigger/.style={decoration={shape start size=.125cm, shape end size=.5cm}},
  smaller/.style={decoration={shape start size=.5cm, shape end size=.125cm}},
  decoration={shape backgrounds,
              shape sep={.25cm, between borders},shape scaled}
}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate, bigger, red!50]   (0,1) -- (3,2);
  \fill [decorate, smaller, blue!50] (0,0) -- (3,1);
\end{tikzpicture}
```

If this key is set to false (which is the default), then only the start width and height are used. Note that the keys `shape width` and `shape height` set the start and end height simultaneously.

/pgf/decoration/shape start width=⟨*length*⟩ (no default, initially `2.5pt`)

The starting width of the shape.

/pgf/decoration/shape start height=⟨*length*⟩ (no default, initially `2.5pt`)

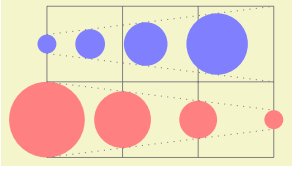
The starting height of the shape.

/pgf/decoration/shape start size=⟨*length*⟩ (style, no default)

Sets both the start height and start width simultaneously.

/pgf/decoration/shape end width=⟨*length*⟩ (no default, initially `2.5pt`)

The recommended ending width of the shape. Note that this is the width that a shape will take only if it is drawn exactly at the end of the path.



```
\usetikzlibrary {decorations.shapes}
\tikzset{
  bigger/.style={decoration={shape start size=.25cm, shape end size=1cm}},
  smaller/.style={decoration={shape start size=1cm, shape end size=.25cm}},
  decoration={shape backgrounds,
              shape sep={.25cm, between borders},shape scaled}
}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \fill [decorate,bigger,
         decoration={shape sep={.25cm, between borders}}, blue!50]
    (0,1.5) -- (3,1.5);
  \fill [decorate,smaller,
         decoration={shape sep={1cm, between centers}},   red!50]
    (0,.5)  -- (3,.5);
  \draw [gray, dotted] (0,1.625) -- (3,2)    (0,1.375) -- (3,1)
                       (0,1)     -- (3,.625) (0,0)     -- (3,.375);
\end{tikzpicture}
```

/pgf/decoration/shape end height=⟨*length*⟩ (no default)

The recommended ending height of the shape.

/pgf/decoration/shape end size=⟨*length*⟩ (style, no default)

Set both the end height and end width simultaneously.

50.7 Text Decorations

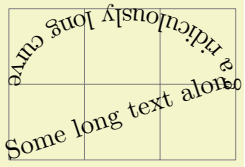
TikZ Library decorations.text

```
\usepgflibrary{decorations.text}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.text]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.text} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.text] % ConTeXt when using TikZ
```

The decoration in this library decorates the path with some text. This can be used to draw text that follows a curve.

Decoration `text along path`

This decoration decorates the path with text. This drawing of the text is a "side effect" of the decoration. The to-be-decorated path is only used to determine where the characters should be put and it is thrown away after the decoration is done. This is why no line is shown in the following example.




```
\usetikzlibrary {decorations.text}
\catcode`\|12
\begin{tikzpicture}[decoration={text along path,
    text={Some long text along a ridiculously long curve that}}]
  \draw [help lines] grid (3,2);
  \draw [decorate] (0,0) -- (3,1) arc (0:180:1.5 and 1);
\end{tikzpicture}
```

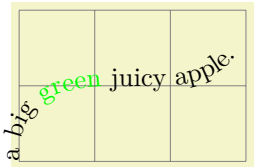
PGF "does its best" to typeset the text, however you should note the following points:

- Each character in the text is typeset in a separate `\hbox`. This means that if you want fancy things like kerning or ligatures you will have to manually annotate the characters in the decoration text within a group, for example, `W{\kern-1ptA}TER`.
- Each character is positioned using the center of its baseline. To move the text vertically (relative to the path), the additional decoration key `raise` should be used (see Section 24.4 for details).
- No attempt is made to ensure characters do not overlap when the angle between segments is considerably less than 180° (this is tricky to do in TEX without a huge processing overhead). In general this should not be too much of a problem, but, once again, kerning can be used in most cases to overcome any undesirable effects.
- It is only possible to typeset text in math mode under considerable restrictions. Math mode is entered and exited using any character of category code 3 (e.g., in plain TEX this is `$`). Math subscripts and superscripts need to be contained within braces (e.g., `{^y_i}`) as do commands like `\times` or `\cdot`. However, even modestly complex mathematical typesetting is unlikely to be successful along a path (or even desirable).
- Some inaccuracies in positioning may be particularly apparent at input segment boundaries. This can (unfortunately) only be solved on a case-by-case basis by individually kerning the offending characters within a group.

The following keys are used by the `text` decoration:

`/pgf/decoration/text=`⟨*text*⟩ (no default, initially empty)

Sets the text to typeset along the curve. Consecutive spaces are ignored, so `\ ` (or `\space` in LATEX) should be used to insert multiple spaces. It is possible to format the text using normal formatting commands, such as `\it`, `\bf` and `\color`, within customizable delimiters. Initially these delimiters are both `|` (however, care will be needed regarding the category codes of delimiters – see below).




```
\usetikzlibrary {decorations.text}
\catcode`\|12
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
          text={a big |\color{green}|green|| juicy apple.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

By following the first delimiter with `+`, the formatting commands are added to any existing formatting.

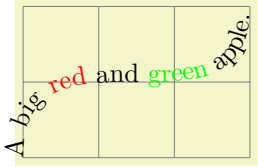



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate,decoration={text along path,
          text={a |\large|big |+\bf\color{red}|red|| juicy apple.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

Internally, the text is stored in the macro `\pgfdecorationtext`. Any characters that have not been typeset when the end of the path has been reached will be stored in `\pgfdecorationrestoftext`.

`/pgf/decoration/text format delimiters={`⟨*before*⟩`}{`⟨*after*⟩`}` (no default, initially `{|}{}`)

Set the characters that the text decoration will use to parse formatting commands. If ⟨*after*⟩ is empty, then ⟨*before*⟩ will be used for both delimiters. In general you should stick to characters whose category codes are `11` or `12`. As `+` is used to indicate that the specified format commands are added to any existing ones, you should avoid using `+` as a delimiter.



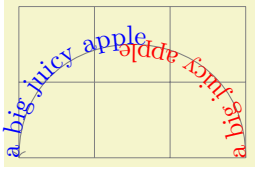

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \path [decorate, decoration={text along path,text format delimiters={[}{]},
           text={A big [\color{red}]red[] and [\color{green}]green[] apple.}}]
    (0,0) .. controls (0,2) and (3,0) .. (3,2);
\end{tikzpicture}
```

`/pgf/decoration/text color=`⟨*color*⟩ (no default, initially `black`)

The color of the text.

`/pgf/decoration/reverse path=`⟨*boolean*⟩ (no default, initially `false`)

This key reverses the path. This is especially useful for typesetting text along different sides of curves.



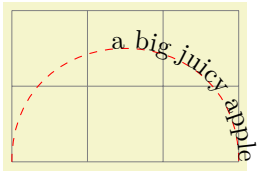

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [gray, ->]
    [postaction={decoration={text along path,
      text={a big juicy apple}, text color=red}, decorate}]
    [postaction={decoration={text along path,
      text={a big juicy apple}, text color=blue, reverse path}, decorate}]
    (3,0) .. controls (3,2) and (0,2) .. (0,0);
\end{tikzpicture}
```

`/pgf/decoration/text align={`⟨*alignment options*⟩`}` (no default)

This changes the key path to `/pgf/decoration/text align` and executes ⟨*alignment options*⟩.

`/pgf/decoration/text align/align=`⟨*alignment*⟩ (no default, initially `left`)

Aligns the text according to ⟨*alignment*⟩, which should be one of `left`, `right`, or `center`.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\draw [red, dashed]
[postaction={decoration={text along path, text={a big juicy apple},
  text align={align=right}}, decorate}]
(0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

`/pgf/decoration/text align/left` (style, no value)

Aligns the text to the left end of the path.

`/pgf/decoration/text align/right` (style, no value)

Aligns the text to the right end of the path.

`/pgf/decoration/text align/center` (style, no value)

Aligns the text to the center of the path.

`/pgf/decoration/text align/left indent=`⟨*length*⟩ (no default, initially `0pt`)

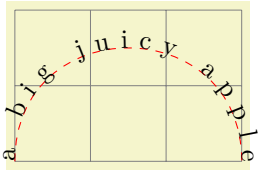
Specifies a distance which the automaton should move along before it starts typesetting the text.

`/pgf/decoration/text align/right indent=`⟨*length*⟩ (no default, initially `0pt`)

Specifies a distance before the end of the path, where the automaton should stop typesetting the text.

/pgf/decoration/text align/fit to path=⟨*boolean*⟩ (no default, initially `false`)

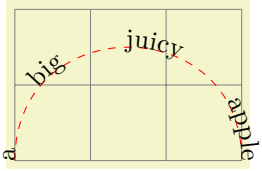
This key makes the decoration automaton try to fit the text to the length of the path. The automaton shifts forward by a small amount between each character in order to fit the text to the path. If, however, the length of the text is longer than the length of the path (i.e., the automaton would have to shift *backwards* between characters) this key will have no effect.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [red, dashed]
    [postaction={decoration={text along path, text={a big juicy apple},
      text align=fit to path}, decorate}]
    (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

/pgf/decoration/text align/fit to path stretching spaces=⟨*boolean*⟩ (no default, initially `false`)

This key works like the previous key except the automaton shifts forward only for space characters (including `\space`, but *excluding* `\ `).




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [red, dashed]
    [postaction={decoration={text along path, text={a big juicy apple},
      text align={fit to path stretching spaces}}, decorate}]
    (0,0) .. controls (0,2) and (3,2) .. (3,0);
\end{tikzpicture}
```

Decoration `text effects along path`

This decoration is similar to the `text along path` decoration except that each character is inserted into the picture as a Ti*k*Z node, and node options (such as `text`, `scale` and `opacity`) can be used to create 'text effects'.




```
\usetikzlibrary {decorations.text,math}
\bfseries\large
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!}, text align=center,
  text effects/.cd,
    character count=\i, character total=\n,
    characters={evaluate={\c=\i/\n*100;}, text along path, text=red!\c!orange},
    character widths={text along path, xslant=0, yscale=1}}]

\path [postaction={decorate}, preaction={decorate,
  text effects={characters/.append={yscale=-1.5, opacity=0.5,
    text=gray, xslant=(\i/\n-0.5)*3}}}]
   (0,0) .. controls ++(2,1) and ++(-2,-1) .. (3,4);
\end{tikzpicture}
```

There are some important differences between this decoration and the `text along path` decoration:

- formatting (e.g., font and color) cannot be specified in the decoration text. They can only be specified using the keys described below.
- as a consequence of using the Ti*k*Z node options, this decoration is only available in Ti*k*Z.
- due to the number of computations involved, this is quite a slow decoration.

The following keys are shared with the `text along path` decoration:

/pgf/decoration/text=`{`⟨*text*⟩`}` (no default)

Set the text this decoration will use. Braces can be used to group multiple characters together, or commands that should not be expanded until they are typeset, for example `gr{\"o}{\ss}eren`. You should *not* use the formatting delimiters or math mode characters that the `text along path` decoration supports.

`/pgf/decoration/text align=`⟨*align*⟩ (no default)

This sets the alignment of the text along the path. The ⟨*align*⟩ argument should be `left`, `right` or `center`. Spreading the text out, or stretching the spaces between words is *not* supported.

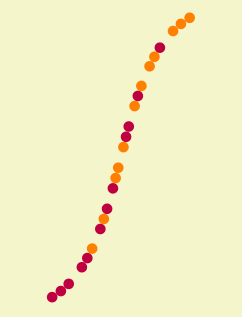
The decoration text can be thought of as consisting of *characters* arranged in to sequences of *letters* to make *words* which are separated by a *word separator*. This, however, does not mean that you are limited to using only natural language as the decoration text.

0 0 0 - 0 0 1 - 0 1 0 - 0 1 1 - 1 0 0 - 1 0 1 - 1 1 0 - 1 1 1

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={000-001-010-011-100-101-110-111},
  text effects/.cd,
    path from text,
    word separator=-,
    every letter/.style={shape=rectangle, fill=blue!20, draw=blue!40}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

In addition, it is possible to replace characters with Ti*k*Z code:



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={000-001-010-011-100-101-110-111}, text align=center,
  text effects/.cd,
    word separator=-,
    replace characters=0 with {\fill [purple] circle [radius=2pt]; },
    replace characters=1 with {\fill [orange] circle [radius=2pt]; },
    replace characters=- with {\path circle [radius=2pt]; },
    every letter/.style={shape=rectangle, fill=blue!20, draw=blue!40}}]

\path [decorate] (0,0) .. controls ++(2,0) and ++(-2,0) .. (3,4);
\end{tikzpicture}
```

There are many keys and styles that can be used to add effects to the decoration text. Many of these keys have the parent path `/pgf/decoration/text effects/`, but for convenience, these keys can be accessed using the following key:

`/tikz/text effects=`{⟨*options*⟩} (no default)

Execute every option in {⟨*options*⟩} with the key path for each option temporarily set to `/pgf/decoration/text effects/`.

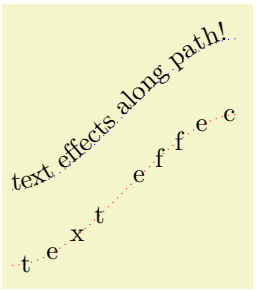
The following keys can be used to customise the appearance of text in the `text effects along path` decoration.

`/pgf/decoration/text effects/every character` (style, no value)

Set the effects that will be applied to every character in the decoration text. The effects will typically be Ti*k*Z node options. Initially, this style is empty so the decoration simply positions nodes at the appropriate position along the path. In order to make the text 'follow the path' like the `text along path` decoration the following key can be added to the `every character` style.

`/pgf/decoration/text effects/text along path` (style, no value)

This style automatically sets the Ti*k*Z keys `transform shape` (to make the character slope with the path), `anchor=baseline` (to make the baseline of the characters 'sit' on the path) and `inner xsep=0pt` (to horizontally fit each node to the character it contains, reducing the spacing between characters).




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
    text={text effects along path!}}]

\path [draw=red, dotted, postaction={decorate}]
  (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\path [draw=blue, dotted, yshift=1cm, postaction={decorate},
  text effects={text along path}]
  (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

`/pgf/decoration/text effects/characters={`⟨*effects*⟩`}` (no default)

Shorthand for the `every character`.

`/pgf/decoration/text effects/character` ⟨*number*⟩ (style, no value)

Specify additional effects for the character ⟨*number*⟩.

`/pgf/decoration/text effects/every letter` (style, no value)

Specify additional effects for every letter (i.e., every character that isn't the word separator) in the decoration text.

`/pgf/decoration/text effects/letter` ⟨*number*⟩ (style, no value)

Specify the effects for letter ⟨*number*⟩ in *every* word.

`/pgf/decoration/text effects/every first letter` (style, no value)

Specify additional effects for the first letter in *every* word.

`/pgf/decoration/text effects/every last letter` (style, no value)

Specify additional effects for the last letter in *every* word.

`/pgf/decoration/text effects/every word` (style, no value)

Specify additional effects for every word in the decoration text.

`/pgf/decoration/text effects/word` ⟨*number*⟩ (style, no value)

Specify additional effects for word ⟨*number*⟩ in the decoration text.

`/pgf/decoration/text effects/word` ⟨*m*⟩ `letter` ⟨*n*⟩ (style, no value)

Specify additional effects for letter ⟨*n*⟩ in word ⟨*m*⟩ in the decoration text.

`/pgf/decoration/text effects/every word separator` (style, no value)

Specify additional effects for every character that is a word separator.

`/pgf/decoration/text effects/word separator=`⟨*character*⟩ (no default, initially `space`)

Specify the character that is to be used as the word separator. This *must* be a single character such as `a` or `-` or the special value `space` (which should be used to indicate that spaces should be used as the separator).

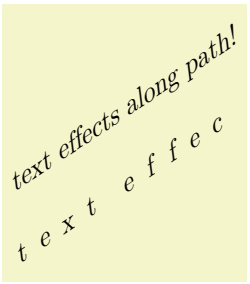
By default, the width for each character is calculated according to the bounding box of the node in which it is contained. However, if the node is rotated or slanted, or has a substantial `inner sep`, this bounding box will be quite big. The following key enables different effects to be applied to the node that is used to calculate the width.

`/pgf/decoration/text effects/every character width` (style, no value)

This style is applied to the (invisible) nodes used for calculating the width of a character node.

`/pgf/decoration/text effects/character widths={`⟨*effects*⟩`}` (no default)

Shorthand for the `every character width` style.



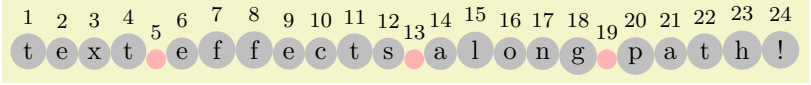

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!}, text align=center,
  text effects/.cd,
    character count=\i,
    characters={xslant=0.5, text along path, name=c-\i}}]

\path [decorate] (0,0) -- (3,2);
\path [decorate,
  text effects={character widths={inner xsep=0pt, xslant=0}}]
  (0,1) -- (3,3);
\end{tikzpicture}
```

It is possible to parameterize effects, perhaps for doing calculations, or labelling nodes based on the number of the character in the decoration text. To access the number of the character, and the total number of characters the following keys can be used. However, these keys should *not* be used inside the style keys given above.

`/pgf/decoration/text effects/character count`=⟨*macro*⟩ (no default)

Store the number of the character being typeset in ⟨*macro*⟩.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text,
    character count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

\path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

`/pgf/decoration/text effects/character total`=⟨*macro*⟩ (no default)

Store the total number of the characters in the decoration text in ⟨*macro*⟩. This key can be used with the `character count` key to produce some quite pleasing effects:

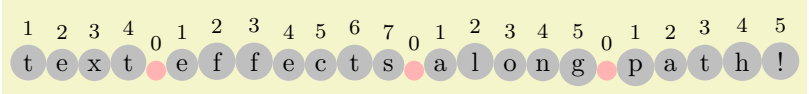



```
\usetikzlibrary {decorations.text,math}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    character count=\i, character total=\n,
    characters={text along path, evaluate={\c=\i/\n*100;},
      text=orange!\c!blue, scale=\i/\n+0.5}}]

\path [decorate]
  (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

`/pgf/decoration/text effects/letter count`=⟨*macro*⟩ (no default)

Store the number of letter being typeset (i.e., the position of the character in the word) in ⟨*macro*⟩. Numbering starts at `1` and the character acting as a word separator is numbered `0`.



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, letter count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

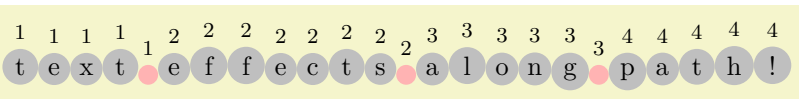
\path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text/effetcs/letter total=⟨*macro*⟩ (no default)

Store the number of letters in the current word in ⟨*macro*⟩. When the character is the word separator, this value is `0`.

/pgf/decoration/text effects/word count=⟨*macro*⟩ (no default)

Store the number of words in the decoration text in ⟨*macro*⟩. Numbering starts at `1`. When the character is the word separator, ⟨*macro*⟩ takes the number of the previous word. If the decoration text starts with a word separator ⟨*macro*⟩ will be `0`.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, word count=\i, every word separator/.style={fill=red!30},
    characters={text along path, shape=circle, fill=gray!50}}]

 \path [decorate, text effects={characters/.append={label=above:\footnotesize\i}}] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/word total=⟨*macro*⟩ (no default)

Store the total number of words in the decoration text in ⟨*macro*⟩.

It is also possible to apply effects to specific characters such as coloring every instance of the character `a`, or changing the font of every `T` in the decoration text:

/pgf/decoration/text effects/style characters={⟨*characters*⟩} with {⟨*effects*⟩} (no default)

This key enables ⟨*effects*⟩ to be applied to every character in the decoration text that is specified in ⟨*characters*⟩.

Falsches Üben von Xylophonmusik quält jeden größeren Zwerg

```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={Falsches {\"U}ben von Xylophonmusik qu{\"a}lt jeden gr{\"o}{\ss}eren Zwerg},
  text effects/.cd,
    path from text,
    style characters=aeiou{\"U}{\"a}{\"o} with {text=blue},
    characters={text along path}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/path from text={⟨*true or false*⟩} (default `true`)

When this key is set to `true` and the decorated path consists only of a single point, the decoration will calculate the width of the decoration text using all the specified parameters as if the decorated path was actually a straight line starting from the given point. This 'virtual' straight line is then decorated with the text.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/path from text angle=⟨*angle*⟩ (no default)

When used in conjunction with the `path from text` key, the straight line that is used as the decorated path is rotated by ⟨*angle*⟩ around the starting point.

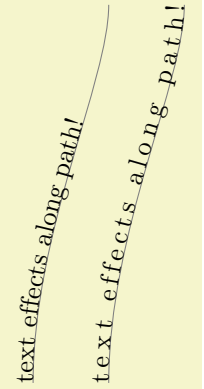



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/fit text to path=⟨*true or false*⟩ (default `true`)

This key will make the decoration increase the space between characters so that the entire path is used by the decoration.

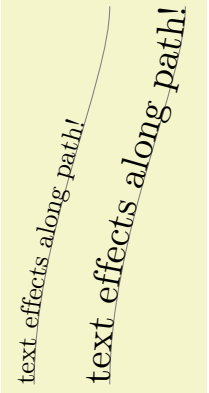



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/every character/.style={text along path}}]

\path [draw=gray, postaction={decorate}, rotate=90]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\path [draw=gray, postaction={decorate}, rotate=90, yshift=-1cm,
  text effects={fit text to path}]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

/pgf/decoration/text effects/scale text to path=⟨*true or false*⟩ (default `true`)

This key will make the decoration scale the text so that the entire path is used by the decoration.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/every character/.style={text along path}}]

\path [draw=gray, postaction={decorate}, rotate=90]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\path [draw=gray, postaction={decorate}, rotate=90, yshift=-1cm,
  text effects={scale text to path}]
  (0,0) .. controls ++(2,0) and ++(-1,0) .. (5,-1);
\end{tikzpicture}
```

/pgf/decoration/text effects/reverse text (no value)

Reverse the order of the characters in the decoration text. This may be useful if using ‘right-to-left‘ languages. Unfortunately, any leading ‘soft’ spaces in the original text will be lost.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    reverse text,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0) .. controls ++(1,0) and ++(-1,0) .. (3,2);
\end{tikzpicture}
```

It is important to note that the `reverse text` key reverses the text *before* doing anything else. This means that the numbering of characters, letters and words will still be in the normal order, so any parameterized effects will have to take this into account. Alternatively, to get the numbering to follow the reversed text, it is possible to reverse the path and then invert the scale:

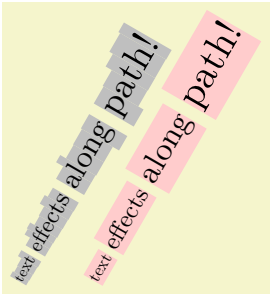



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate, text effects={reverse text}] (0,0);
\path [blue, decorate, decoration={reverse path},
    text effects={characters/.append={scale=-1}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/group letters (no value)

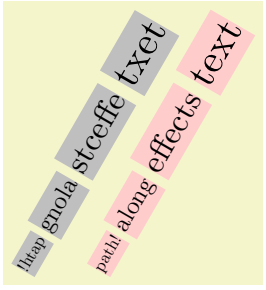
Group sequences of letters together so they are treated as a single ‘character’.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    every word separator/.style={fill=none},
    character count=\i, character total=\n,
    characters={text along path, fill=gray!50, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\path [decorate, text effects={group letters,
  characters/.append={fill=red!20}}]
  (1,0);
\end{tikzpicture}
```

The order in which the `reverse text` and `group letters` keys are applied is important:

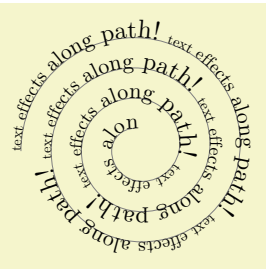



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    every word separator/.style={fill=none},
    character count=\i, character total=\n,
    characters={text along path, fill=gray!50, scale=\i/\n+0.5}}]

\path [decorate, text effects={reverse text, group letters}] (0,0);
\path [decorate, text effects={group letters, reverse text,
  characters/.append={fill=red!20}}] (1,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/repeat text=⟨*times*⟩ (no default)

Usually, when the decoration runs out of text, it simply stops. This key will make the decoration repeat the decoration text for the specified number of ⟨*times*⟩. If no value is given the text will be repeated until the path is finished. There are two points to remember however. Firstly the numbering of characters, letters and words will be restarted each time the text is repeated. Secondly, the options for alignment, scaling or fitting the text to the path, fitting the path to the text, and so on, are computed using the decoration text before the decoration starts. If any of these options are given the behavior of the `repeat text` key is undefined, but typically it will be ignored.

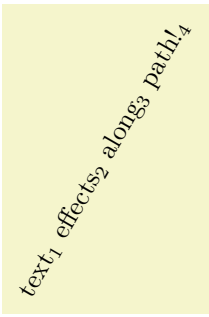



```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!\ },
  text effects/.cd,
    repeat text,
    character count=\m, character total=\n,
    characters={text along path, scale=0.5+\m/\n/2}}]

\path [draw=gray, ultra thin, postaction=decorate]
  (180:2) \foreach \a in {0,...,12}{ arc (180-\a*90:90-\a*90:1.5-\a/10) };
\end{tikzpicture}
```

/pgf/decoration/text effects/character command=⟨*macro*⟩ (no default)

This key specifies a command that is executed when each character is placed in the node. The ⟨*macro*⟩ should be an ordinary TEX macro which takes one argument. The argument will be a macro which when expanded will contain the current character.

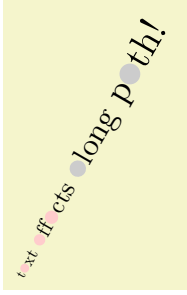



```
\usetikzlibrary {decorations.text}
\def\mycommand#1{#1$_\n$}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60, group letters,
    word count=\n,
    every word/.style={character command=\mycommand},
    characters={text along path}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

/pgf/decoration/text effects/replace characters=⟨*characters*⟩ with {⟨*code*⟩} (no default)

Replace the node for each character in ⟨*characters*⟩ with ⟨*code*⟩. The ⟨*code*⟩ can be thought of as describing a little picture or marking which will be used instead of the character node. The origin will be the current point along the decoration path. Any transformations associated with the ⟨*characters*⟩ (e.g., applied with the `every character` or `every letter` styles) will also be applied to ⟨*code*⟩.




```
\usetikzlibrary {decorations.text}
\begin{tikzpicture}[decoration={text effects along path,
  text={text effects along path!},
  text effects/.cd,
    path from text, path from text angle=60,
    replace characters=e with {\fill [red!20]   (0,1mm) circle [radius=1mm];},
    replace characters=a with {\fill [black!20] (0,1mm) circle [radius=1mm];},
    character count=\i, character total=\n,
    characters={text along path, scale=\i/\n+0.5}}]

\path [decorate] (0,0);
\end{tikzpicture}
```

50.8 Fractal Decorations

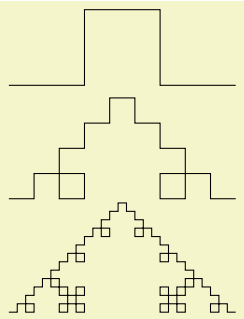
TikZ Library `decorations.fractals`

```
\usepgflibrary{decorations.fractals}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[decorations.fractals]  % ConTeXt when using pure PGF
\usetikzlibrary{decorations.fractals} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[decorations.fractals] % ConTeXt when using TikZ
```

The decorations of this library can be used to create fractal lines. To use them, you typically have to apply the decoration repeatedly to an originally straight path.

Decoration Koch curve type 1

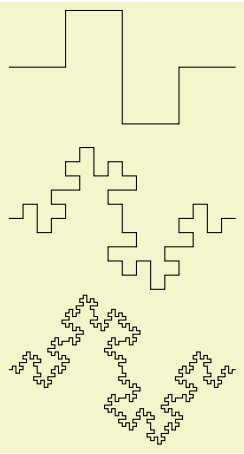
This decoration replaces a straight line by a "rectangular bump". By repeatedly applying this replacement, different levels of the Koch curve fractal can be created. Its Hausdorff dimension is $\log 5 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch curve type 1]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1.5) -- (3,-1.5) }};
  \draw decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}};
\end{tikzpicture}
```

Decoration Koch curve type 2

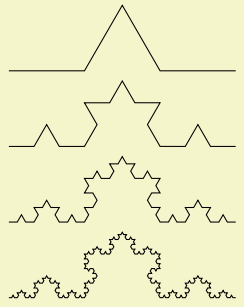
This decoration replaces a straight line by a "rectangular sine". Its Hausdorff dimension is $3/2$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch curve type 2]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-2) -- (3,-2) }};
  \draw decorate{ decorate{ decorate{ (0,-4) -- (3,-4) }}};
\end{tikzpicture}
```

Decoration Koch snowflake

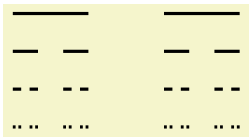
This decoration replaces a straight line by a "line with a spike". The Hausdorff dimension of Koch's snowflake's is $\log 4 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Koch snowflake]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-1) -- (3,-1) }};
  \draw decorate{ decorate{ decorate{ (0,-2) -- (3,-2) }}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-3) -- (3,-3) }}}};
\end{tikzpicture}
```

Decoration Cantor set

This decoration replaces a straight line by a “line with a gap in the middle”. The Hausdorff dimension of the Cantor set is $\log 2 / \log 3$.



```
\usetikzlibrary {decorations.fractals}
\begin{tikzpicture}[decoration=Cantor set,very thick]
  \draw decorate{ (0,0) -- (3,0) };
  \draw decorate{ decorate{ (0,-.5) -- (3,-.5) }};
  \draw decorate{ decorate{ decorate{ (0,-1) -- (3,-1) }}};
  \draw decorate{ decorate{ decorate{ decorate{ (0,-1.5) -- (3,-1.5) }}}};
\end{tikzpicture}
```

51 Entity-Relationship Diagram Drawing Library

TikZ Library er

```
\usetikzlibrary{er} % LaTeX and plain TeX
\usetikzlibrary[er] % ConTeXt
```

This packages provides styles for drawing entity-relationship diagrams.

This library is intended to help you in creating E/R-diagrams. It defines only few new styles, but using the style `entity` instead of saying `rectangle,draw` makes the code more expressive.

51.1 Entities

The package defines a simple style for drawing entities:

/tikz/entity (style, no value)

This style is to be used with nodes that represent entity types. It causes the node's shape to be set to a rectangle that is drawn and whose minimum size and width are set to sensible values.

Note that this style is called `entity` despite the fact that it is to be used for nodes representing entity *types* (the difference between an entity and an entity type is the same as the difference between an object and a class in object-oriented programming). If this bothers you, feel free to define a style `entity type` instead.




```
\usetikzlibrary {er,positioning}
\begin{tikzpicture}
  \node[entity] (sheep)                    {Sheep};
  \node[entity] (genome) [right=of sheep] {Genome};
\end{tikzpicture}
```

/tikz/every entity (style, no value)

This style is evoked by the style `entity`. To change the appearance of entities, you can change this style.



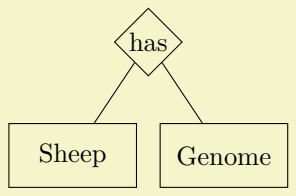

```
\usetikzlibrary {er,positioning}
\begin{tikzpicture}
  [every entity/.style={draw=blue!50,fill=blue!20,thick}]
  \node[entity] (sheep)                    {Sheep};
  \node[entity] (genome) [right=of sheep] {Genome};
\end{tikzpicture}
```

51.2 Relationships

Relationships are drawn using styles that are very similar to the styles for entities.

/tikz/relationship (style, no value)

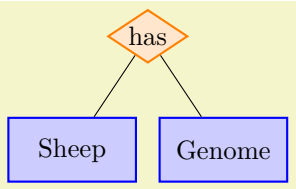
This style works like `entity`, only it is to be used for relationships. Again, `relationship`s are actually relationship types.




```
\usetikzlibrary {er}
\begin{tikzpicture}
  \node[entity] (sheep)  at (0,0)   {Sheep};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

/tikz/every relationship (style, no value)

Works like `every entity`.

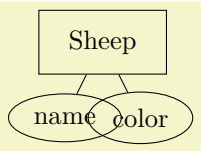



```
\usetikzlibrary {er}
\begin{tikzpicture}
  [every entity/.style={fill=blue!20,draw=blue,thick},
   every relationship/.style={fill=orange!20,draw=orange,thick,aspect=1.5}]
  \node[entity] (sheep)  at (0,0)   {Sheep};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

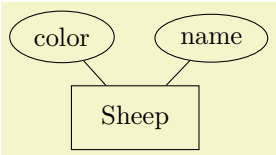
51.3 Attributes

/tikz/attribute (style, no value)

This style is used to indicate that a node is an attribute. To connect an attribute to its entity, you can use, for example, the `child` command or the `pin` option.




```
\usetikzlibrary {er}
\begin{tikzpicture}
  \node[entity] (sheep)  {Sheep}
    child {node[attribute] {name}}
    child {node[attribute] {color}};
\end{tikzpicture}
```



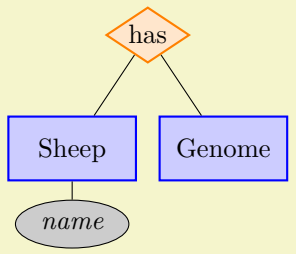

```
\usetikzlibrary {er}
\begin{tikzpicture}[every pin edge/.style=draw]
  \node[entity,pin={[attribute]60:name},pin={[attribute]120:color}] {Sheep};
\end{tikzpicture}
```

/tikz/key attribute (style, no value)

This style is intended for key attributes. By default, the will cause the attribute to be typeset in italics. Typically, underlining is used instead, but that looks ugly and it is difficult to implement in TEX.

/tikz/every attribute (style, no value)

This style is used with every attribute, and therefore also for every key attribute.




```
\usetikzlibrary {er}
\begin{tikzpicture}
  [text depth=1pt,
   every attribute/.style={fill=black!20,draw=black},
   every entity/.style={fill=blue!20,draw=blue,thick},
   every relationship/.style={fill=orange!20,draw=orange,thick,aspect=1.5}]

  \node[entity] (sheep)  at (0,0)   {Sheep}
    child {node  [key attribute] {name}};
  \node[entity] (genome) at (2,0)   {Genome};
  \node[relationship]    at (1,1.5) {has}
    edge (sheep)
    edge (genome);
\end{tikzpicture}
```

52 Externalization Library

by Christian Feuersänger

Ti*k*Z Library external

```
\usetikzlibrary{external} % LaTeX and plain TeX
\usetikzlibrary[external] % ConTeXt
```

This library provides a high-level automatic or semi-automatic export feature for Ti*k*Z pictures. Its purpose is to convert each picture to a separate PDF without changing the document as such.

It also externalizes `\label` information (and other aux file related stuff) using auxiliary files.

52.1 Overview

There are several reasons why external images for at least some pictures are of interest:

1. Larger picture require a considerable amount of time, which is necessary for every compilation. However, only few images will change from run to run. It can simply save time to export finished images and include them as final graphics.
2. It may be desirable to have final images for some graphics, for example to include them in third–party programs or to communicate them electronically.
3. It may be necessary to typeset a file in environments where PGF and Ti*k*Z are not available. In this case, external images are the only way to ensure compatibility.

The purpose of this library is to provide a way to export any Ti*k*Z-picture to separate PDF (or `eps`) images without changing the main document. It is actually a simple user interface to the `\beginpgfgraphicnamed` ... `\endpgfgraphicnamed` framework of PGF which is discussed in section 111.

52.2 Requirements

For most users, the library does not need special attention since requirements are met anyway. It collects all tokens between `\begin{tikzpicture}` and the next following `\end{tikzpicture}` and replaces them by the appropriate graphics or it takes steps to generate such an image.

It can't expand macros during this step, so the only requirement is that every picture's end is directly reachable from its beginning, without further macro expansion. Furthermore, the library assumes that all LaTeX pictures are ended with `\end{tikzpicture}`.

The library always searches for the *next* picture's end, `\end{tikzpicture}`. As a consequence, you can't use nested pictures directly. You *can* nest pictures, but you have to avoid that the nested picture's `\end` command is found before the outer `\end` command (for example using bracing constructs or by writing the nested picture into a separate macro call).

Consider using the `\tikzexternaldisable` method in case you'd like to skip selected pictures which do not meet the requirements.

52.3 A Word About ConTeXt And Plain TeX

Currently, the basic layer backend `\beginpgfgraphicnamed` ... `\endpgfgraphicnamed` relies on LaTeX only, so externalization is currently only supported for LaTeX.

52.4 Externalizing Graphics

After loading the library, a call to `\tikzexternalize` is necessary to activate the externalization.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize % activate!

\begin{document}
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}

A simple image is \tikz \fill (0,0) circle[radius=5pt];.
\end{document}
```

The method works as follows: if the document is typeset normally, the library searches for replacement images for every picture. Filenames are generated automatically in the default configuration. In our case, the two file names will be `main-figure0` and `main-figure1`. If they exist, those images are simply included and the pictures as such are not processed. If graphics files do not exist, steps are taken to generate the missing ones. Since (currently) only one output file can be set, each missing image needs to be generated by a separate run of LaTeX in which the `\jobname` is set to the desired image file name. In the default configuration `mode=convert with system call`, these commands are issued automatically by using the `\write18` method to call system commands. It is also possible to output every required file name or to generate a `makefile`; users will need to issue the required commands manually (or with `make`). The probably most comfortable way is to use the default configuration with

```
pdflatex -shell-escape main
```

which authorizes `pdflatex` to call itself recursively to generate the images. When it finishes, all images are generated and the document already includes them.

From this point on, successive runs of LaTeX will use the final graphics files, the pictures won't be used anymore. Section 52.5 contains details about how to submit such a file to environments where PGF is not available.

`\tikzexternalize`[⟨*optional arguments*⟩]

This command activates the externalization. It installs commands to replace every TikZ-picture. It needs to be called before `\begin{document}` because it may need to install its separate shipout routine.

The ⟨*optional arguments*⟩ can be any of the keys described below.

Note that the generation/modification of auxiliary files like `.aux`, `.toc` etc. is usually suppressed while a single image is externalized (details for `\label` support follow).

It is also possible to write `\tikzexternalize{`⟨*main job name*⟩`}` if the argument is delimited by curly braces. This case is mainly for backwards compatibility and is no longer necessary. Since it might be useful in rare circumstances, it is documented in section 52.4.5.

A detailed description about the process of externalization is provided in section 52.4.5.

`\tikzexternalrealjob`

After the library is loaded, this macro will *always* contain the correct main job's name (in the example above, it is `main`). It is to be used instead of `\jobname` when the externalization is in effect.

`\pgfactualjobname`

Once `\tikzexternalize` has been called, `\pgfactualjobname` contains the name of the currently generated output file (which may be `main` or `main-figure0` or `main-figure1` in our example above).

`\jobname`

The value of `\jobname` is one of `\tikzexternalrealjob` or `\pgfactualjobname`, depending on the configuration. In short: if auxiliary file support (`\label` and `\ref`) is activated, `\jobname=\tikzexternalrealjob` (since that's the base file name of auxiliary files).

/tikz/external/system call={⟨*template*⟩} (no default)

A template string used to generate system calls. Inside of {⟨*template*⟩}, the macro \image can be used as placeholder for the image which is about to be generated while \texsource contains the main file name (in truth, it contains \input{⟨*main file name*⟩}, but that doesn't matter).

The default depends on the value of \pgfsysdriver. For pgfsys-pdftex.def, it is

```
\tikzset{external/system call={pdflatex \tikzexternalcheckshellescape -halt-on-error
    -interaction=batchmode -jobname "\image" "\texsource"}}
```

where \tikzexternalcheckshellescape inserts the value of the configuration key shell escape if and only if the current document has been typeset with -shell-escape[6].

Other drivers result in slightly different calls. There is support for lualatex, xelatex, and dvips. The precise values are written to the .log file as soon as you attempt to compile a document.

The argument {⟨*template*⟩} will be expanded using \edef, so any control sequences will be expanded. During this evaluation, '\\' will result in a normal backslash, '\'. Furthermore, double quotes '"', single quotes ''', semicolons and dashes '-' will be made to normal characters if any package uses them as macros. This ensures compatibility with the german package, for example.

/tikz/external/shell escape={⟨*command-line arg*⟩} (no default, initially -shell-escape)

Contains the command line option for latex which enables the \write18 feature. For TEX-Live, this is -shell-escape. For MiKTEX, you should use \tikzexternalize[shell escape=-enable-write18].

52.4.1 Support for Labels and References In External Files

The external library comes with extra support for \label and \ref (and other commands which usually store information in the .aux file) inside an external files.

In particular, it supports the two use-cases

a) \ref to something in the main document inside an externalized graphics or

b) \label in the externalized graphics which is referenced in the main document.

The only restriction is that you need to compile your document multiple times (as usual for references).

NOTE: support for a) is unavailable for versions up to and including PGF 3.0.1.

/tikz/external/aux in dpth={⟨*boolean*⟩} (no default, initially true)

Allows to enable or disable the feature which handles references and labels as part of image externalization. Disabling it will safe one \newwrite command, i.e. a write register.

Also see the disable dependency files feature.

Here are some implementation details on how references within/from external graphics work for those who would like to know the details:

For point a), a \ref inside of an externalized graphics works by reading the main document's .aux file. To this end, the standard mode=convert with system call detects such references and reschedules the externalization to \end{document}.[7] Other values of mode require just one attempt to externalize the picture.

Note that \pageref is not supported (sorry).

Point b) works as follows: a \label inside of an externalized graphics causes the external library to generate separate auxiliary files for every external image. These files are called ⟨*imagename*⟩.dpth. The extension .dpth indicates that the file also contains the image's depth (the baseline key of Ti*k*Z). Furthermore, anything which would have been written to an .aux file will be redirected to the .dpth file – but only things which occur inside of the externalized tikzpicture environment. When the main document loads the image, it will copy the .dpth file into the main .aux file. Then, successive compilations of the main document contain the external \label information. In other words, a \label in an external graphics needs the following work flow:

[6] Note that this is always true for the default configuration. This security consideration applies mainly for mode=list and make which will also work *without* shell escapes.

[7] Note that this requires the atveryend package. The purpose to reschedule the externalization is to access the main job's aux file, but only after it has been written completely.

1. The external graphics needs to be generated together with its `.dpth` (usually automatically by Ti*k*Z).
2. The main document includes the external graphics and copies the `.dpth` content into its main `.aux` file.
3. The main document needs to be translated once again to re-read its `.aux` file[8].

This does also work if a `\label`/`\ref` combination is implemented itself by a `tikzpicture` (a feature offered by `pgfplots`).

52.4.2 Customizing the Generated File Names

The default filename for externalized graphics is '⟨*real file name*⟩`-figure_`⟨*number*⟩' where ⟨*number*⟩ ranges from 0 to whatever is required. However, there are a couple of ways to change the generated filenames:

- Changing the overall file name using a `prefix`,
- Changing the file name for a single figure using `\tikzsetnextfilename`,
- Changing the file name for a restricted set of figures using `figure name`.

`/tikz/external/prefix=`{⟨*file name prefix*⟩} (no default, initially `empty`)

A shortcut for `\tikzsetexternalprefix`{⟨*file name prefix*⟩}, see below.

`\tikzsetexternalprefix`{⟨*file name prefix*⟩}

Assigns a common prefix used by all file names. For example,

```
\tikzsetexternalprefix{figures/}
```

will prepend `figures/` to every external graphics file name.

Please note that `\tikzsetexternalprefix` is the *only* way to assign a prefix in case you want to prepare your document for environments where PGF is not installed (see section 52.5).

`\tikzsetnextfilename`{⟨*file name*⟩}

Sets the file name for the *next* Ti*k*Z picture or `\tikz` short command. It will *only* be used for the next picture.

Pictures for which no explicit file name has been set (or the next file name is empty) will get automatically generated file names.

Please note that `prefix` will still be prepended to {⟨*file name*⟩}.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize[prefix=figures/] % activate

\begin{document}

\tikzsetnextfilename{trees}
\begin{tikzpicture} % will be written to 'figures/trees.pdf'
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}

\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle[radius=5pt];. % will be written to 'figures/simple.pdf'

\begin{tikzpicture} % will be written to 'figures/main-figure0.pdf'
   \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

[8] Note that it is not possible to activate the content of an auxiliary file after `\begin{document}` in LaTeX.

```
pdflatex -shell-escape main
```

/tikz/external/figure name={⟨*name*⟩} (no default)

Same as \tikzsetfigurename{⟨*name*⟩}.

\tikzsetfigurename{⟨*name*⟩}

Changes the names of *all* following figures. It is possible to change figure name during the document either using \tikzset{external/figure name={⟨*name*⟩}} or with this command. A unique counter will be used for each different {⟨*name*⟩}, and each counter will start at 0.

The value of prefix will be applied after figure name has been evaluated.

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize % activate

\begin{document}

\begin{tikzpicture} % will be written to 'main-figure0.pdf'
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}

{
  \tikzsetfigurename{subset_}
  A simple image is \tikz \fill (0,0) circle[radius=5pt];. % will be written to 'subset_0.pdf'

  \begin{tikzpicture} % will be written to 'subset_1.pdf'
     \draw[help lines] (0,0) grid (5,5);
  \end{tikzpicture}
}% here, the old file name will be restored:

\begin{tikzpicture} % will be written to 'main-figure1.pdf'
   \draw (0,0) -- (5,5);
\end{tikzpicture}
\end{document}
```

The scope of figure name ends with the next closing brace.

Remark: Use \tikzset{external/figure name/.add={prefix_}{_suffix_}} to add a prefix_ and a _suffix_ to the actual value of figure name.

\tikzappendtofigurename{⟨*suffix*⟩}

Appends ⟨*suffix*⟩ to the actual value of figure name.

It is a shortcut for \tikzset{external/figure name/.add={}{⟨*suffix*⟩}} (a shortcut which is also supported if Ti*k*Z is not installed, see below).

52.4.3 Remaking Figures or Skipping Figures

\tikzpicturedependsonfile{⟨*file name*⟩}

Adds a dependency for the *next* picture which is about to be externalized. If the command is invoked within a picture environment, it adds a dependency for the surrounding picture. Dependencies are written into ⟨*target file*⟩.dep in the format

⟨*target file*⟩.\tikzexternalimgextension: ⟨*file name*⟩.

The effect is that if ⟨*file name*⟩ changes, the external graphics associated with the picture shall be remade.

This command uses the contents of \tikzexternalimgextension to check for graphics. If you encounter difficulties with image extensions, consider redefining this macro (after \tikzexternalize).

Limitations: this command is currently only supported for `mode=list and make` and the generated `makefile`.

`\tikzexternalfiledependsonfile{`⟨*external graphics*⟩`}{`⟨*file name*⟩`}`

A variant of `\tikzpicturedependsonfile` which adds a dependency for an ⟨*external graphics*⟩. The argument ⟨*external graphics*⟩ must be the path as it would have been generated by the `external` library, i.e. without file extension but including any prefixes.

`/tikz/external/disable dependency files` (no value)

Allows to (irreversibly) disable the generation of file dependencies. Disabling it will safe one `\newwrite` command, i.e. a write register. Note that the write register is only allocated if the feature has been used at all. This key needs to be provided as argument to `\tikzexternalize` (or it needs to be set before calling `\tikzexternalize`).

Also see the `aux in dpth` key.

`/tikz/external/force remake={`⟨*boolean*⟩`}` (default `true`)

A boolean which is used to customize the up-to-date checks of all following figures. Every up-to-date check will fail, resulting in automatic regeneration of every following figure.

```
\tikzset{external/force remake}
\begin{tikzpicture}
    \draw (0,0) circle[radius=5pt];
\end{tikzpicture}
```

You can also use `force remake` inside of a local TEX group to remake only selected pictures. The example

```
\tikz \draw (0,0) -- (1,1);

{
\tikzset{external/force remake}
\begin{tikzpicture}
   \draw (0,0) circle[radius=5pt];
\end{tikzpicture}
}

\tikz \draw (0,0) -- (1,1);
```

will only apply `force remake` to the second figure.

Up-to-date checks are applied for `mode=convert with system call` and the makefile generated by `mode=list and make`.

`/tikz/external/remake next={`⟨*boolean*⟩`}` (default `true`)

A variant of `force remake` which applies only to the next image.

`/tikz/external/export next={`⟨*boolean*⟩`}` (default `true`)

A boolean which can be used to disable the export mechanism for single pictures.

`/tikz/external/export={`⟨*boolean*⟩`}` (no default, initially `true`)

A boolean which can be used to disable the export mechanism for all pictures inside of the current TEX-scope.

```
\begin{document}
\begin{tikzpicture} % will be exported
    ...
\end{tikzpicture}

{
\tikzset{external/export=false}
\begin{tikzpicture} % won't be exported
    ...
\end{tikzpicture}
...
}

\begin{tikzpicture} % will be exported
    ...
\end{tikzpicture}
\end{document}
```

For LaTeX, the feature lasts until the next `\end{⟨·⟩}` (this holds for every call to `\tikzset`).

`/tikz/external/up to date check={⟨choice⟩}` (no default, initially `md5`)

The `external` lib has to decide when some existing figure is up-to-date. In such a case, it can be used without remaking it. Outdated pictures will be remade.

The key `up to date check` allows to choose among a couple of heuristics which are supposed to catch the most important reasons to remake a figure.

The `up to date check` can be overrule by any of the `force remake` or `remake next` keys: if one of them is true, the figure is not up-to-date.

The choice simple is based on the existence of the file: the file is up-to-date if and only if it exists.

The choice md5 generates an MD5 checksum of the picture for which the up-to-date check is running. The MD5 is compared against the MD5 of the previous run, which, in turn, will be written into an extra file with the extension `.md5`. This file will be modified if and only if the MD5 comparison indicates a difference. The MD5 computation is based on the pdfTeX method `\pdfmdfivesum`. If it is unavailable for some reason, the choice `diff` will be used instead.

The choice diff is the same as MD5 – except that it compares the picture content as-is instead of a hash. The `.md5` file will be used to compare an old version with the current one – but its content is some "normalized" version of the picture for internal use.

Attention: the content–based strategies `md5` and `diff` operate on the picture content – and only on the picture content. Here, "picture content" only includes the top–level tokens; no expansion is applied and no included files are part of the strategies. If you change preamble styles, you have to rebuild the figures manually (for example by deleting the generated graphics files). If you have include files, consider using `\tikzpicturedependsonfile` and its variants. Since this key provides heuristics, you should always remake your figures before you finally publish your document. Example: Suppose we have the following picture which depends on a command `\mycommand`:

```
\def\mycommand{My comment}

\begin{tikzpicture}

\node at (0,0) {\mycommand};

\end{tikzpicture}
```

What happens if you change "My comment" to "My super comment"? Well, `external` will *not* pick it up; you will need to handle this manually. However, if you modify anything between `\begin{tikzpicture}` and `\end{tikzpicture}`, the `external` library *will* pick it up and regenerate the picture.

The `up to date check` is applied for `mode=convert with system call` and `mode=list and make`.

`\tikzexternaldisable`

Allows to disable the complete externalization. While `export next` will still collect the contents of picture environments, this command uninstalls the hooks for the `external` library completely. Thus,

nested picture environments or environments where `\end{tikzpicture}` is not directly reachable won't produce compilation failures – although it is not possible to externalize them automatically.

The externalization remains disabled until the end of the next TEX group (or environment) or until the next call to `\tikzexternalenable`.

`\tikzexternalenable`

Re-enables a previously running externalization after `\tikzexternaldisable`.

52.4.4 Customizing the Externalization

`/tikz/external/figure list={`⟨*boolean*⟩`}` (no default, initially `true`)

A boolean which configures whether a figure list shall be generated. A figure list is an output file named `{`⟨*jobname*⟩`}.figlist` which is filled with file names of each figure, one per line.

This file is not used by TEX anymore, its purpose is to issue the required conversion commands `pdflatex -jobname {`⟨*picture file name*⟩`} {`⟨*main file*⟩`}` manually (or in a script). See section 52.4.5 for the details about the expected system call (or activate `mode=convert with system call` and inspect your log file).

```
\documentclass{article}
% main document, called main.tex
\usepackage{tikz}

\usetikzlibrary{external}
\tikzexternalize[
   mode=graphics if exists,
   figure list=true,
   prefix=figures/]

\begin{document}

\tikzsetnextfilename{trees}
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}

\tikzsetnextfilename{simple}
A simple image is \tikz \fill (0,0) circle[radius=5pt];.

\begin{tikzpicture}
   \draw[help lines] (0,0) grid (5,5);
\end{tikzpicture}
\end{document}
```

```
pdflatex main
```

generates `main.figlist` containing

```
figures/trees
figures/simple
figures/main-figure0
```

`/tikz/external/mode={`⟨*choice*⟩`}` (no default, initially `convert with system call`)

Configures what to do with Ti*k*Z pictures (unless we are currently externalizing one particular image, in that case, these modes are ignored).

The preconfigured mode `convert with system call` checks whether external graphics files are up-to-date and includes them if that is the case. Any picture which is not up-to-date will be generated automatically using a system call. The system call can be configured using the `system call` template. The up-to-date check is applied according to the `up to date check` key. As soon as

`convert with system call` is set, the `figure list` will be disabled – such a file is not required. In case you still need or want it, you can enable it after setting `mode`.

Please note that system calls may be disabled for security reasons. For pdflatex, they can be enabled using

```
pdflatex -shell-escape
```

while other TEX variants may need other switches. The feature is sometimes called `\write18`.

The choice `only graphics` always tries to replace pictures with external graphics. It is an error if the graphics file does not exist.

The choice `no graphics` (or, equivalently, `only pictures`) typesets Ti*k*Z pictures without checking for external graphics.

A mixture is `graphics if exists`, it checks whether a suitable graphics file exists and includes it if that is the case. If it does not exist, the picture is typeset using TEX.

Mode `list only` skips every Ti*k*Z picture; it only generates the file `{`⟨*main file*⟩`}.figlist` containing file names for every picture, the contents of any picture environment is thrown away and a replacement text is shown. This implies `figure list=true`. See also the `list and make` mode which includes available graphics.

The mode `list and make` is similar to `list only`: it generates the same file `{`⟨*main file*⟩`}.figlist`, but any images which exist already are included as graphics instead of ignoring them. Furthermore, this mode generates an additional file: `{`⟨*main file*⟩`}`.makefile. This allows to use a work flow like

```
% step 1: generate main.makefile:
pdflatex main
% step 2: generate ALL graphics on 2 processors:
make -j 2 -f main.makefile
% step 3: include the graphics:
pdflatex main
```

This last make method is optional: `list and make` just assumes that images are generated somehow (not necessarily with the generated makefile). The generated makefile allows parallel externalization of graphics on multi-core systems and it supports any file dependencies configured with `\tikzpicturedependsonfile`. Furthermore, it respects the `force remake` and `remake next` keys.

`/tikz/external/verbose IO=`{⟨*boolean*⟩} (no default, initially `true`)

A boolean which configures whether I/O operations shall be listed in the logfile.

`/tikz/external/verbose optimize=`{⟨*boolean*⟩} (no default, initially `true`)

A boolean which configures whether optimization operations shall be listed in the logfile.

`/tikz/external/verbose=`{⟨*boolean*⟩} (no default, initially `true`)

Sets all verbosity flags to ⟨*boolean*⟩.

`/tikz/external/optimize=`{⟨*boolean*⟩} (no default, initially `true`)

Configures whether the conversion process shall be optimized. This affects only the case when `\jobname` differs from the main file name, i.e. when single pictures are converted.

In that case, the main file is compiled as usual – but everything except the selected picture is thrown away. If optimization is enabled, all other pictures won't be processed at all. Furthermore, expensive commands which do not contribute to the selected picture will be thrown away as well.

The default implementation discards `\includegraphics` commands which are *not* inside of the selected picture to reduce conversion time.

It is possible to add commands which shall be optimized away, see below.

`/tikz/external/optimize command away=`⟨\|*command*⟩`{`⟨*required argument count*⟩`}` (no default)

Installs commands to optimize ⟨\|*command*⟩ away. As is described above, optimization applies to the case when single pictures are converted: one usually doesn't need to process (probably expensive) commands which do not contribute to the selected picture.

The argument `{`⟨*required argument count*⟩`}` is either empty or a non-negative integer between 0 and 9. It denotes the number of arguments which should be consumed after ⟨\|*command*⟩. In any case, one

argument in square brackets after the command will be recognized as well. To be more precise, the following cases for arguments of ⟨*command*⟩ are supported:

1. If {⟨*required argument count*⟩} is empty (the default), ⟨*command*⟩ may take one optional argument in square brackets and one in curly braces (which is also optional).
2. If {⟨*required argument count*⟩} is not empty, {⟨*command*⟩} may take one optional argument in square brackets. Furthermore, it expects exactly {⟨*required argument count*⟩} following arguments.

Example:

```
\tikzset{external/optimize command away=\includegraphics}
```

```
\newcommand{\myExpensiveMacro}[1]{Very expensive!}

\tikzset{external/optimize command away=\myExpensiveMacro}
```

```
\newcommand{\myExpensiveMacroWithThreeArgs}[3]{Very expensive!}

\tikzset{external/optimize command away={\myExpensiveMacroWithThreeArgs}{3}}
```

```
% A command with optional argument:
\newcommand{\aFurtherExample}[3][]{Very expensive!}

% consume only two arguments: the first optional one will be processed
% anyway:
\tikzset{external/optimize command away={\myExpensiveMacroWithThreeArgs}{2}}
```

The argument ⟨*command*⟩ must be the name of a single macro. Any occurrence of this macro, together with its arguments, will be removed.

```
\begin{tikzpicture}
    % this picture is currently converted!
\end{tikzpicture}

This here is outside of the converted picture and contains \myExpensiveMacro. It will be discarded.

This call: \myExpensiveMacro[argument=value]{Argument} as well.
And this here: \myExpensiveMacro{Argument} also.
```

The default is to optimize `\includegraphics` away.

This key is actually a style which sets the `optimize/install` and `optimize/restore` keys.

`/tikz/external/optimize/install` (no value)

A command key which contains code to install optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

`/tikz/external/optimize/restore` (no value)

A command key which contains code to undo optimizations. You can append code here (or clear the macro) if you need to modify the optimization.

`/tikz/external/only named`={⟨*boolean*⟩} (no default, initially `false`)

If enabled, only pictures for which file names have been set explicitly using `\tikzsetnextfilename` will be considered, no file names will be generated automatically.

`/pgf/images/include external` (initially `\pgfimage{#1}`)

This command key constitutes the public interface to exchange the `\includegraphics` command used for the image inclusion. If can be overwritten using `include external/.code`={⟨*TEX code*⟩}.

Its description can be found in the corresponding basic layer documentation on page 1182.

Just one example here: you can use

```
\pgfkeys{/pgf/images/include external/.code={\includegraphics[viewport=0 0 211.28 175.686]{#1}}}
```

to manually change the viewport (bounding box) for included graphics.

Another example (of probably limited use) is

```
\pgfkeys{/pgf/images/include external/.code={\href{file:#1}{\pgfimage{#1}}}}
```

which will generate a clickable hyperlink around the image. Clicking on it opens the single exported file[9].

If you want to limit the effects of this key to just one externalized figure, use

```
{
  \pgfkeys{/pgf/images/include external/.code={\includegraphics[viewport=0 0 211.28 175.686]{#1}}}
  \begin{tikzpicture}
    ...
  \end{tikzpicture}
}% this brace ends the effect of `include external'
```

`\tikzifexternalizing{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

This command can be used to check whether an image is currently written to its separate graphics file (if the "grab" procedure is running). If so, the `{`⟨*true code*⟩`}` will be executed. If not, that means if the main document is being typeset normally, the `{`⟨*false code*⟩`}` will be invoked.

This command must be used *after* `\tikzexternalize`.

`\tikzifexternalizingnext{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

Like `\tikzifexternalizing`, but this variant also checks if the next following figure is the one which is about to be written to its separate graphics file.

52.4.5 Details About The Process

The standard run `pdflatex` ⟨*main document*⟩ causes the `external` library to check every occurrence of `\begin{tikzpicture}` and every `\tikz` short command. If it finds a picture which shall be exported, it queries the respective file name and checks whether the file exists already. If so, it includes the external graphics. If not, it requires an externalization which can be done automatically (the default), semi-automatically (with `mode=list and make`) or manually (by issuing the requires system calls somehow).

The library can detect whether it runs in "conversion mode", i.e. if it should only process a single image. To do so, it checks whether the internal macro `\tikzexternalrealjob` exists. If so, its contents is assumed to be ⟨*main document*⟩ (without the suffix `.tex`). Usually, this macro is set by the conversion system call,

```
pdflatex -jobname "main-figure0" "\def\tikzexternalrealjob{main}\input{main}"
```

where `main-figure0` is the picture we are currently externalizing and `main.tex` is the main document.

As soon as "conversion mode" has been detected, PGF changes the output routine. The complete file `main.tex` is processed as normal, but only the part of the desired picture will be written to the output file, in our case `main-figure0.pdf`. The rest of the document is silently thrown away. Of course, such a conversion process is quite expensive since we need to do it for every picture. Since everything except the current picture is thrown away, the library skips all other pictures. Furthermore, any `\includegraphics` commands which are outside of the converted TikZ-picture will be skipped as well. Thus, the conversion process should be much faster than typesetting the complete document, but it still requires its time. Eventually, the call `\input{main}` returns and the picture is ready. From this point on, the external graphics will be used.

There is another possibility to communicate ⟨*main document*⟩ to the subprocess performing the externalization: namely to write '`\tikzexternalize{main}`' into the document. In this case, the conversion system call will be

```
pdflatex -jobname "main-figure0" "main"
```

and the contents of `\tikzexternalrealjob` is set automatically. This case is detected by `\tikzexternalize`, and the `system call` is updated automatically (by patching its `\texsource` template argument). It is not necessary to change the `system call` manually.

The sequence in which system calls are performed and the decision whether they are issued automatically is governed by the `mode` key, consult its documentation for details.

[9] This requires all external graphics files in the same base directory as the main `.pdf` file.

52.5 Using External Graphics Without PGF Installed

Given that every picture has been exported correctly, one may want to compile a file without PGF and Ti*k*Z installed. Ti*k*Z comes with a minimal package which contains just enough commands to replace every `tikzpicture` environment and the `\tikz` short command with the appropriate external graphics. It can be found at

```
latex/pgf/utilities/tikzexternal.sty
```

and needs to be used instead of `\usepackage{tikz}`. So, we comment `\usepackage{tikz}` and `\usetikzlibrary{external}`, load packages `graphicx` and `tikzexternal`, and finally our example from the beginning becomes

```
\documentclass{article}
% main document, called main.tex
%\usepackage{tikz}

\usepackage{graphicx}
\usepackage{tikzexternal}

%\usetikzlibrary{external}
\tikzexternalize

\begin{document}
\begin{tikzpicture}
  \node {root}
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}

A simple image is \tikz \fill (0,0) circle[radius=5pt];.

Furthermore, we might want to draw \tikz[baseline]\draw (0,-1) rectangle (1,1);
\end{document}
```

where the following files are necessary to compile the document:

```
tikzexternal.sty
main.tex
main-figure0.pdf
main-figure1.pdf
main-figure2.pdf
```

If there are any '`.dpth`' files, for example `main-figure2.dpth`, these files are also required. They contain information for the Ti*k*Z `baseline` option (or `\label`s inside external graphics).

Just copy the `.sty` file into the directory of your `main.tex` file and use it as part of your document.

Please keep in mind, that only `tikzpicture` environments and `\tikz` short images are available within the externalization framework. Additionally, calls to `\tikzset` and `\pgfkeys` won't lead to compilation errors because they are simply ignored. But since `pgfkeys` is not available, any option supplied to `\tikzexternalize` is *ignored*.

Attention: Since the simple replacement `\usepackage{tikzexternal}` doesn't support the key–value interface, you *need* to use `\tikzsetexternalprefix` instead of the `prefix` option and `\tikzsetfigurename` instead of the `figure name` option since `\tikzset` is not available in such a context.

Remark: Some of the features of this library are mainly useful to improve the speed of successive document compilations. In other words: you can't use all features in this context, keep it simple.

52.6 `eps` Graphics Export

It is also possible to use `eps` graphics instead of PDF files. There are different ways to produce them, for example to use `pdflatex` and call `pdftops -eps {`⟨*pdf file*⟩`} {`⟨*eps file*⟩`}` afterwards. You could add this command to the `system call` option.

Alternatively, you can use `latex` and `dvips` for image conversion as is explained for the `system call` option, see page 671. See the documentation for the basic level externalization in section 111 for restrictions of other drivers.

52.7 Bitmap Graphics Export

Occasionally, you may have an extremely large graphics which takes long times to render. It might be interesting to generate a bitmap (raster) image, which displays much faster (for example in a presentation). I have used this feature to speed-up the display of large shadings.

The `external` library can be customized to export bitmap images – with the help of external programs. Due to the dependence of external programs, you may need to adjust these commands manually. For example, on my computer, the ImageMagick Suite is installed which comes with the `convert` tool. Together with `pdflatex`, I can define the following style:

```
\tikzset{
    % Defines a custom style which generates BOTH, .pdf and .png export
    % but prefers the .png on inclusion.
    %
    % This style is not pre-defined, you may need to copy-paste and
    % adjust it.
    png export/.style={
        external/system call/.append=
            {; convert -density 300 -transparent white "\image.pdf" "\image.png"},
        %
        /pgf/images/external info,
        /pgf/images/include external/.code={%
            \includegraphics
                [width=\pgfexternalwidth,height=\pgfexternalheight]
                {##1.png}%
        },
    }
}
```

The example above defines a new style called '`png export`' which, when it is set with `\tikzset{png export}` somewhere in the document, modifies the configuration for both file generation and file input. The file generation is modified by appending the ImageMagick command to `system call` (separated by '`;`' as usual on Linux). This is, in principle, enough to generate a `.png` file. The `include external` command is overwritten such that it uses the `.png` file instead of the `.pdf` file (which exists as well in the configuration above). But since a `.png` file can have a much higher resolution than the desired image dimensions, we have to add `width` and `height` explicitly. Usually, the `external` library does not provide size information (it is unnecessary for `.pdf` or `.eps` since these formats have their bounding box information). To enable size information, the style uses the `external info` key, which, in turn, provides the `\pgfexternalwidth` and `\pgfexternalheight` commands.

Now we can use `\tikzset{png export}` either document-wide or just for one particular image. The configuration remains in effect until the end of the current environment (or until the next closing curly brace '`}`').

`/pgf/images/external info=`{⟨*boolean*⟩} (no default, initially `false`)

If this key is activated, the size for any externalized image will be stored explicitly into the associated `.dpth` file.

When the file is included by `\pgfincludeexternalgraphics` (or automatically by the `external` library), the width is available as `\pgfexternalwidth` and the height as `\pgfexternalheight`.

52.8 Compatibility Issues

52.8.1 References In External Pictures

It is allowed if a picture contains references, for example `\tikz \node {Reference to \ref{a:label}};`.

There is just one issue: if the main job is currently compiling, its `.aux` file is not in its final state (even worse: it may not be readable at all). The picture externalization, however, needs the main `.aux` file to query any references.

Thus, you *will* need to invoke `pdflatex -jobname` ⟨*image*⟩ ⟨*mainfile*⟩ *manually* for any image which contains references.

This problem arises only for `mode=convert with system call`. In this case, the `external` library creates a special `\jobname.auxlock` file to check whether the main `.aux` file is currently usable.

52.8.2 Compatibility With Other Libraries or Packages

The `external` library has the following compatibility issues:

1. The `external` library comes with special support for `\usetikzlibrary{fadings}`: the `fadings` library may define local pictures which would be externalized (although they shouldn't). There is special handling to suppress this bug if `\tikzexternalize` is called *after* `\usetikzlibrary{fadings}` or if all fadings are defined *before* `\tikzexternalize`.

2. Problems have been reported when using `\tikzexternalize` (or the basic layer externalization) together with `\usepackage{glossary}`. This problem disappears if `\tikzexternalize` is called *before* `\usepackage{glossary}`.

3. Problems with `\usepackage{pdfpages}` and `\usepackage{vmargin}`: The `external` library replaces the current shipout routine of TEX during its externalization. This might raise problems with other packages which also manipulate the shipout routine (like the mentioned ones). To fix those problems, use

   ```
   \usetikzlibrary{external}

   \tikzifexternalizing{%
       % don't include package XYZ here
   }{%
       \usepackage{pdfpages}
       \usepackage{vmargin}
       ...
   }%
   ```

   This uses the requested packages for the main document, but not for the single, exported graphics.

In general, the `\tikzifexternalizing` feature might be used to solve package conflicts and the `\tikzexternaldisable` and `\tikzexternalenable` features can be used to solve problems with single pictures.

52.8.3 Compatibility With Bounding Box Restrictions

Bounding box restrictions provide no problem when used with `eps` graphics. However, they pose problems for `pdflatex`, so you may need to use the `latex/dvips` combination if you use bounding box restrictions and externalization. Currently, the only possibility for bounding box restrictions and `pdflatex` is to use a combination of `trim left/trim right/baseline`: these keys do not *really* truncate the bounding box, they only store horizontal and vertical shifts (also see the `trim lowlevel` key in this context).

52.8.4 Interoperability With The Basic Layer Externalization

This library is fully compatible with `\beginpgfgraphicnamed`...`\endpgfgraphicnamed` environments. However, you will need to use the `export next=false` key to avoid conflicts:

```
\beginpgfgraphicnamed{picture4}
\tikzset{external/export next=false}
\begin{tikzpicture}
   \draw (0,0) -- (4,4);
\end{tikzpicture}
\endpgfgraphicnamed
```

Please keep in mind that file prefixes do not apply to the basic layer.

53 Fading Library

Ti*k*Z Library fadings

```
\usepgflibrary{fadings}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[fadings]  % ConTeXt when using pure PGF
\usetikzlibrary{fadings} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[fadings] % ConTeXt when using TikZ
```

The package defines a number of fadings, see Section 23 for an introduction. The Ti*k*Z version defines special Ti*k*Z commands for creating fadings. These commands are explained in Section 23.

Fading name	*Example (solid blue faded on checkerboard)*
`west`	
`east`	
`north`	
`south`	
`circle with fuzzy edge 10 percent`	
`circle with fuzzy edge 15 percent`	
`circle with fuzzy edge 20 percent`	
`fuzzy ring 15 percent`	

54 Fitting Library

TikZ Library fit

```
\usetikzlibrary{fit} % LaTeX and plain TeX
\usetikzlibrary[fit] % ConTeXt
```

The library defines (currently only two) options for fitting a node so that it contains a set of coordinates.

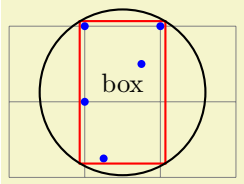
When you load this library, the following options become available:

`/tikz/fit=`⟨*coordinates or nodes*⟩ (no default)

This option must be given to a `node` path command. The ⟨*coordinates or nodes*⟩ should be a sequence of TikZ coordinates or node names, one directly after the other without commas (like with the `plot coordinates` path operation). Examples are `(1,0) (2,2)` or `(a) (1,0) (b)`, where `a` and `b` are nodes.

For this sequence of coordinates, a minimal bounding box is computed that encompasses all the listed ⟨*coordinates or nodes*⟩. For coordinates in the list, the bounding box is guaranteed to contain this coordinate, for nodes it is guaranteed to contain the `east`, `west`, `north` and `south` anchors of the node. In principle (the details will be explained in a moment), things are now set up such that the text box of the node will be exactly this bounding box.

Here is an example: We fit several points in a rectangular node. By setting the `inner sep` to zero, we see exactly the text box of the node. Then we fit these points again in a circular node. Note how the circle encompasses exactly the same bounding box.




```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt,thick,
                    dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[dot] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};

  \node[draw=red,   fit=(a) (b) (c) (d) (e)] {box};
  \node[draw,circle,fit=(a) (b) (c) (d) (e)] {};
\end{tikzpicture}
```

Every time the `fit` option is used, the following style is also applied to the node:

`/tikz/every fit` (style, initially empty)

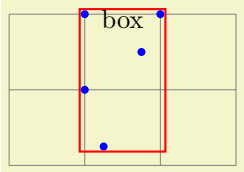
Set this style to change the appearance of a node that uses the `fit` option.

The exact effects of the `fit` option are the following:

1. A minimal bounding box containing all coordinates is computed. Note that if a coordinate like `(a)` is used that contains a node name, this has the same effect as explicitly providing the `(a.north)` and `(a.south)` and `(a.west)` and `(a.east)`. If you wish to refer only to the center of the `a` node, use `(a.center)` instead.
2. The `text width` option is set to the width of this bounding box.
3. The `align=center` option is set.
4. The `anchor` is set to `center`.
5. The `at` position of the node is set to the center of the computed bounding box.
6. After the node has been typeset, its height and depth are adjusted such that they add up to the height of the computed bounding box and such that the text of the node is vertically centered inside the box.

The above means that, generally speaking, if the node contains text like `box` in the above example, it will be centered inside the box. It will be difficult to put the text elsewhere, in particular, changing the `anchor` of the node will not have the desired effect. Instead, what you should do is to create a node with the `fit` option that does not contain any text, give it a name, and then use normal nodes to add text at the desired positions. Alternatively, consider using the `label` or `pin` options.

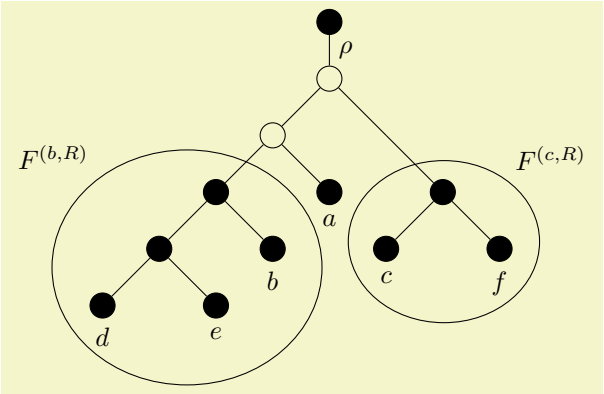
Suppose, for instance, that in the above example we want the word “box” to appear inside the box, but at its top. This can be achieved as follows:




```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt,thick,
                    dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[dot] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};

  \node[draw=red,fit=(a) (b) (c) (d) (e)] (fit) {};
  \node[below] at (fit.north) {box};
\end{tikzpicture}
```

Here is a real-life example that uses fitting:



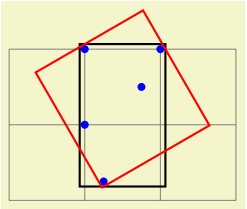

```
\usetikzlibrary {fit,shapes.geometric}
\begin{tikzpicture}
  [vertex/.style={minimum size=2pt,fill,draw,circle},
   open/.style={fill=none},
   sibling distance=1.5cm,level distance=.75cm,
   every fit/.style={ellipse,draw,inner sep=-2pt},
   leaf/.style={label={[name=#1]below:$#1$}},auto]

  \node [vertex] (root) {}
  child { node [vertex,open] {}
    child { node [vertex,open] {}
      child { node [vertex] (b's parent) {}
        child { node [vertex] {}
          child { node [vertex,leaf=d] {} }
          child { node [vertex,leaf=e] {} } }
        child { node [vertex,leaf=b] {} } }
      child { node [vertex,leaf=a] {} } }
    child { node [coordinate] {}
      child[missing]
      child { node [vertex] (f's parent) {}
        child { node [vertex,leaf=c] {} }
        child { node [vertex,leaf=f] {} } } }
    edge from parent node {$\rho$} };

  \node [fit=(d) (e) (b) (b's parent),label=above left:$F^{(b,R)}$] {};
  \node [fit=(c) (f) (f's parent),label=above right:$F^{(c,R)}$]    {};
\end{tikzpicture}
```

`/tikz/rotate fit=`⟨*angle*⟩ (no default, initially 0)

This key fits ⟨*coordinates or nodes*⟩ inside a node that is rotated by ⟨*angle*⟩. As a side effect, it also sets the `/tikz/rotate` key.



```
\usetikzlibrary {fit}
\begin{tikzpicture}[inner sep=0pt,thick,
  dot/.style={fill=blue,circle,minimum size=3pt}]
  \draw[help lines] (0,0) grid (3,2);
  \node[dot] (a) at (1,1) {};
  \node[dot] (b) at (2,2) {};
  \node[dot] (c) at (1,2) {};
  \node[dot] (d) at (1.25,0.25) {};
  \node[dot] (e) at (1.75,1.5) {};
  \node[draw, fit=(a) (b) (c) (d) (e)] {};
  \node[draw=red, rotate fit=30, fit=(a) (b) (c) (d) (e)] {};
\end{tikzpicture}
```

55 Fixed Point Arithmetic Library

Ti*k*Z Library fixedpointarithmetic

```
\usepgflibrary{fixedpointarithmetic}  % LaTeX and plain TeX when using pure pgf
\usepgflibrary[fixedpointarithmetic]  % ConTeXt when using pure pgf
\usetikzlibrary{fixedpointarithmetic} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[fixedpointarithmetic] % ConTeXt when using TikZ
```

This library provides an interface to the LaTeX package `fp` for fixed point arithmetic. In addition to loading this library you must ensure `fp` is loaded, otherwise errors will occur.

55.1 Overview

Whilst the mathematical engine that comes with pgf is reasonably fast and flexible when it comes to parsing, the accuracy tends to be fairly low, particularly for expressions involving many operations chained together. In addition the range of values that can be computed is very small: ±16383.99999. Conversely, the `fp` package has a reasonably high accuracy, and can perform computations over a wide range of values (approximately $\pm 9.999 \times 10^{17}$), but is comparatively slow and not very flexible, particularly regarding parsing.

This library enables the combination of the two: the flexible parser of the pgf mathematical engine with the evaluation accuracy of `fp`. There are, however, a number of important points to bear in mind:

- Whilst `fp` supports very large numbers, pgf and Ti*k*Z do not. It is possible to calculate the result of `2^20` or `1.2e10+3.4e10`, but it is not possible to use these results in pictures directly without some "extra work".
- The pgf mathematical engine will still be used to evaluate lengths, such as `10pt` or `3em`, so it is not possible for an length to exceed the range of values supported by TeX-dimensions (±16383.99999pt), even though the resulting expression is within the range of `fp`. So, for example, one can calculate `3cm*10000`, but not `3*10000cm`.
- Not all of the functions listed in Section 94, have been mapped onto `fp` equivalents. Of those that have been, it is not guaranteed that functions will perform in the same way as they do in pgf. Reference should be made to the documentation for `fp`.
- In pgf, trigonometric functions such as `sin` and `cos` assume arguments are in degrees, and functions such as `asin` and `acos` return results in degrees. Although `fp` uses radians for such functions, pgf automatically converts arguments from degrees to radians, and converts results from radians to degrees, to ensure everything "works properly".
- The overall speed will actually be slower than using pgf mathematical engine. The calculating power of `fp` comes at the cost of an increased processing time.

55.2 Using Fixed Point Arithmetic in PGF and Ti*k*Z

The following key is provided to use `fp` in pgf and Ti*k*Z:

/pgf/fixed point arithmetic=⟨*options*⟩ (no default)

alias `/tikz/fixed point arithmetic`

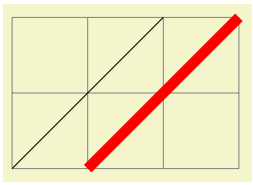
This key will set the key path to `/pgf/fixed point`, and execute ⟨*options*⟩. Then it will install the necessary commands so that the pgf parser will use `fp` to perform calculations. The best way to use this key is as an argument to a scope or picture. This means that `fp` does not always have to be used, and pgf can use its own mathematical engine at other times, which can lead to a significant reduction in the time for a document to compile.

Currently there are only a few keys key supported for ⟨*options*⟩:

/pgf/fixed point/scale results=⟨*factor*⟩ (no default)

As noted above, `fp` can process a far greater range of numbers than pgf and Ti*k*Z. In order to use results from `fp` in a `{pgfpicture}` or a `{tikzpicture}` they need to be scaled. When this key is used pgf will scale results of any evaluation by ⟨*factor*⟩. However, as it is not desirable for every part of every expression to be scaled, scaling will only take place if a special prefix `*` is used. If `*` is used at

the beginning of an expression the evaluation of the expression will evaluated and then multiplied by ⟨*factor*⟩.



```
\usepgflibrary {fixedpointarithmetic}
\begin{tikzpicture}[fixed point arithmetic={scale results=10^-6}]
\draw [help lines] grid (3,2);
\draw (0,0) -- (2,2);
\draw [red, line width=4pt] (*1.0e6,0) -- (*3.0e6,*2.0e6);
\end{tikzpicture}
```

A special case of scaling involves plots of data containing large numbers from files. It is possible to "pre-process" a file, typically using the application that generates the data, to either precede the relevant column with `*` or to perform the scaling as part of the calculation process. However, it may be desirable for the data in a plot to appear in a table as well, so, two files would be required, one pre-processed for plotting, and one not. This extra work may be undesirable so the following keys are provided:

`/pgf/fixed point/scale file plot x`=⟨*factor*⟩ (no default)

This key will scale the first column of data read from a file before it is plotted. It is independent of the `scale results` key.

`/pgf/fixed point/scale file plot y`=⟨*factor*⟩ (no default)

This key will scale the second column of data read from a file before it is plotted.

`/pgf/fixed point/scale file plot z`=⟨*factor*⟩ (no default)

This key will scale the third column of data read from a file before it is plotted.

56 Floating Point Unit Library

by Christian Feuersänger

Ti*k*Z Library fpu

```
\usepgflibrary{fpu}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[fpu]  % ConTeXt when using pure PGF
\usetikzlibrary{fpu} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[fpu] % ConTeXt when using TikZ
```

The floating point unit (fpu) allows the full data range of scientific computing for use in PGF. Its core is the PGF math routines for mantissa operations, leading to a reasonable trade-off between speed and accuracy. It does not require any third-party packages or external programs.

56.1 Overview

The fpu provides a replacement set of math commands which can be installed in isolated placed to achieve large data ranges at reasonable accuracy. It provides at least[10]the IEEE double precision data range, $-1 \cdot 10^{324}, \ldots, 1 \cdot 10^{324}$. The absolute smallest number bigger than zero is $1 \cdot 10^{-324}$. The FPU's relative precision is at least $1 \cdot 10^{-4}$ although operations like addition have a relative precision of $1 \cdot 10^{-6}$.

Note that the library has not really been tested together with any drawing operations. It should be used to work with arbitrary input data which is then transformed somehow into PGF precision. This, in turn, can be processed by PGF.

56.2 Usage

/pgf/fpu={⟨*boolean*⟩} (default `true`)

This key installs or uninstalls the FPU. The installation exchanges any routines of the standard math parser with those of the FPU: `\pgfmathadd` will be replaced with `\pgfmathfloatadd` and so on. Furthermore, any number will be parsed with `\pgfmathfloatparsenumber`.

1Y2.0e0]

```
\usepgflibrary {fpu}
\pgfkeys{/pgf/fpu}
\pgfmathparse{1+1}\pgfmathresult
```

The FPU uses a low-level number representation consisting of flags, mantissa and exponent[11].To avoid unnecessary format conversions, `\pgfmathresult` will usually contain such a cryptic number. Depending on the context, the result may need to be converted into something which is suitable for PGF processing (like coordinates) or may need to be typeset. The FPU provides such methods as well.

Use `fpu=false` to deactivate the FPU. This will restore any change. Please note that this is not necessary if the FPU is used inside of a TeX group – it will be deactivated afterwards anyway.

It does not hurt to call `fpu=true` or `fpu=false` multiple times.

Please note that if the `fixedpointarithmetic` library of PGF will be activated after the FPU, the FPU will be deactivated automatically.

/pgf/fpu/output format=float|sci|fixed (no default, initially `float`)

This key allows to change the number format in which the FPU assigns `\pgfmathresult`.

The predefined choice `float` uses the low-level format used by the FPU. This is useful for further processing inside of any library.

1Y2.17765411e23]

```
\usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=float}
\pgfmathparse{exp(50)*42}\pgfmathresult
```

The choice `sci` returns numbers in the format ⟨*mantissa*⟩`e`⟨*exponent*⟩. It provides almost no computational overhead.

[10]To be more precise, the FPU's exponent is currently a 32-bit integer. That means it supports a significantly larger data range than an IEEE double precision number – but if a future TeX version may provide low-level access to doubles, this may change.

[11]Users should *always* use high level routines to manipulate floating point numbers as the format may change in a future release.

5.6154816e14

```
\usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=sci}
\pgfmathparse{4.22e-8^-2}\pgfmathresult
```

The choice `fixed` returns normal fixed point numbers and provides the highest compatibility with the PGF engine. It is activated automatically in case the FPU scales results.

0.000000999985

```
\usepgflibrary {fpu}
\pgfkeys{/pgf/fpu,/pgf/fpu/output format=fixed}
\pgfmathparse{sqrt(1e-12)}\pgfmathresult
```

`/pgf/fpu/scale results={`⟨*scale*⟩`}` (no default)

A feature which allows semi-automatic result scaling. Setting this key has two effects: first, the output format for *any* computation will be set to `fixed` (assuming results will be processed by PGF's kernel). Second, any expression which starts with a star, `*`, will be multiplied with ⟨*scale*⟩.

`/pgf/fpu/scale file plot x={`⟨*scale*⟩`}` (no default)
`/pgf/fpu/scale file plot y={`⟨*scale*⟩`}` (no default)
`/pgf/fpu/scale file plot z={`⟨*scale*⟩`}` (no default)

These keys will patch PGF's `plot file` command to automatically scale single coordinates by ⟨*scale*⟩.

The initial setting does not scale `plot file`.

`\pgflibraryfpuifactive{`⟨*true-code*⟩`}{`⟨*false-code*⟩`}`

This command can be used to execute either ⟨*true-code*⟩ or ⟨*false-code*⟩, depending on whether the FPU has been activated or not.

`/pgf/fpu/install only={`⟨*list of names*⟩`}` (no default)

Unfortunately, the FPU is currently incompatible with drawing operations. However, it can still be useful to replace single definitions with FPU counterparts to avoid errors of the kind `Dimension too large` which tend to happen when transformation matrices are inverted.

This key allows to specify a list of definitions to be pulled into the current scope. *Note that there is no reverse operation to uninstall these definitions at the moment*, so it is advisable to do this in a group. Conveniently, Ti*k*Z paths form an implicit group, so you can use this key on a path as well.

You have to be aware of the limitations that the FPU imposes. It will not magically give TEX better precision, but it will avoid overflow or underflow situations for large or small operands by rescaling them. In the following example, in the first case the FPU variant performs much better than the normal variant, however, in the second case where a rescaling would not in fact be needed the rescaling introduces a small round-off error.

10000.000000(good)2.9999000000000(bad)

```
\usepgflibrary {fpu}
\begingroup
\pgfkeys{/pgf/fpu/install only={divide}}
\pgfmathparse{12.34/0.001234}\pgfmathresult (good)
\pgfmathparse{12/4}\pgfmathresult (bad)
\endgroup
```

This key is introduced in PGFv3.1.6 and marked stable since PGFv3.1.8.

56.3 Comparison to the fixed point arithmetic library

There are other ways to increase the data range and/or the precision of PGF's math parser. One of them is the `fp` package, preferable combined with PGF's `fixedpointarithmetic` library. The differences between the FPU and `fp` are:

- The FPU supports at least the complete IEEE double precision number range, while `fp` covers only numbers of magnitude $\pm 1 \cdot 10^{17}$.
- The FPU has a uniform relative precision of about 4–5 correct digits. The fixed point library has an absolute precision which may perform good in many cases – but will fail at the ends of the data range (as every fixed point routines does).

- The FPU has potential to be faster than `fp` as it has access to fast mantissa operations using PGF's math capabilities (which use TEX registers).

56.4 Command Reference and Programmer's Manual

56.4.1 Creating and Converting Floats

`\pgfmathfloatparsenumber{`⟨*x*⟩`}`

Reads a number of arbitrary magnitude and precision and stores its result into `\pgfmathresult` as floating point number $m \cdot 10^e$ with mantissa and exponent base 10.

The algorithm and the storage format is purely text-based. The number is stored as a triple of flags, a positive mantissa and an exponent, such as

```
\pgfmathfloatparsenumber{2}
\pgfmathresult
```

1Y2.0e0]

Please do not rely on the low-level representation here, use `\pgfmathfloattomacro` (and its variants) and `\pgfmathfloatcreate` if you want to work with these components.

The flags encoded in `\pgfmathresult` are represented as a digit where '0' stands for the number $\pm 0 \cdot 10^0$, '1' stands for a positive sign, '2' means a negative sign, '3' stands for 'not a number', '4' means $+\infty$ and '5' stands for $-\infty$.

The mantissa is a normalized real number $m \in \mathbb{R}$, $1 \leq m < 10$. It always contains a period and at least one digit after the period. The exponent is an integer.

Examples:

Flags: 0; Mantissa 0.0; Exponent 0.

```
\pgfmathfloatparsenumber{0}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.0; Exponent -1.

```
\pgfmathfloatparsenumber{0.2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 4.2; Exponent 1.

```
\pgfmathfloatparsenumber{42}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 2.05; Exponent 3.

```
\pgfmathfloatparsenumber{20.5E+2}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 1.0; Exponent 6.

```
\pgfmathfloatparsenumber{1e6}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

Flags: 1; Mantissa 5.21513; Exponent -11.

```
\pgfmathfloatparsenumber{5.21513e-11}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E.
```

The argument ⟨*x*⟩ may be given in fixed point format or the scientific "e" (or "E") notation. The scientific notation does not necessarily need to be normalized. The supported exponent range is (currently) only limited by the TEX-integer range (which uses 31 bit integer numbers).

`/pgf/fpu/handlers/empty number={`⟨*input*⟩`}{`⟨*unreadable part*⟩`}` (no default)

This command key is invoked in case an empty string is parsed inside of `\pgfmathfloatparsenumber`. You can overwrite it to assign a replacement `\pgfmathresult` (in float!).

The initial setting is to invoke `invalid number`, see below.

`/pgf/fpu/handlers/invalid number={`⟨*input*⟩`}{`⟨*unreadable part*⟩`}` (no default)

This command key is invoked in case an invalid string is parsed inside of `\pgfmathfloatparsenumber`. You can overwrite it to assign a replacement `\pgfmathresult` (in float!).

The initial setting is to generate an error message.

`/pgf/fpu/handlers/wrong lowlevel format={`⟨*input*⟩`}{`⟨*unreadable part*⟩`}` (no default)

This command key is invoked whenever `\pgfmathfloattoregisters` or its variants encounter something which is not a properly formatted low-level floating point number. As for `invalid number`, this key may assign a new `\pgfmathresult` (in floating point) which will be used instead of the offending ⟨*input*⟩.

The initial setting is to generate an error message.

`\pgfmathfloatqparsenumber{`⟨*x*⟩`}`

The same as `\pgfmathfloatparsenumber`, but does not perform sanity checking.

`\pgfmathfloattofixed{`⟨*x*⟩`}`

Converts a number in floating point representation to a fixed point number. It is a counterpart to `\pgfmathfloatparsenumber`. The algorithm is purely text based and defines `\pgfmathresult` as a string sequence which represents the floating point number ⟨*x*⟩ as a fixed point number (of arbitrary precision).

Flags: 1; Mantissa 5.2; Exponent -4→0.00052

```
\pgfmathfloatparsenumber{0.00052}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
$\to$
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

Flags: 1; Mantissa 1.23456; Exponent 6→1234560.00000000

```
\pgfmathfloatparsenumber{123.456e4}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
$\to$
\pgfmathfloattofixed{\pgfmathresult}
\pgfmathresult
```

`\pgfmathfloattoint{`⟨*x*⟩`}`

Converts a number from low-level floating point representation to an integer (by truncating the fractional part).

123456

```
\pgfmathfloatparsenumber{123456}
\pgfmathfloattoint{\pgfmathresult}
\pgfmathresult
```

See also `\pgfmathfloatint` which returns the result as float.

`\pgfmathfloattosci{`⟨*float*⟩`}`

Converts a number from low-level floating point representation to scientific format, 1.234*e*4. The result will be assigned to the macro `\pgfmathresult`.

\pgfmathfloatvalueof{⟨*float*⟩}

Expands a number from low-level floating point representation to scientific format, 1.234*e*4.

Use \pgfmathfloatvalueof in contexts where only expandable macros are allowed.

\pgfmathfloatcreate{⟨*flags*⟩}{⟨*mantissa*⟩}{⟨*exponent*⟩}

Defines \pgfmathresult as the floating point number encoded by ⟨*flags*⟩, ⟨*mantissa*⟩ and ⟨*exponent*⟩. All arguments are characters and will be expanded using \edef.

Flags: 1; Mantissa 1.0; Exponent 327

```
\pgfmathfloatcreate{1}{1.0}{327}
\pgfmathfloattomacro{\pgfmathresult}{\F}{\M}{\E}
Flags: \F; Mantissa \M; Exponent \E
```

\pgfmathfloatifflags{⟨*floating point number*⟩}{⟨*flag*⟩}{⟨*true-code*⟩}{⟨*false-code*⟩}

Invokes ⟨*true-code*⟩ if the flag of ⟨*floating point number*⟩ equals ⟨*flag*⟩ and ⟨*false-code*⟩ otherwise.

The argument ⟨*flag*⟩ can be one of

0 to test for zero,

1 to test for positive numbers,

+ to test for positive numbers,

2 to test for negative numbers,

- to test for negative numbers,

3 for "not-a-number",

4 for $+\infty$,

5 for $-\infty$.

It's not zero!It's positive!It's not negative!It's positive!It's not negative!

```
\usetikzlibrary {fpu}
\pgfmathfloatparsenumber{42}
\pgfmathfloatifflags{\pgfmathresult}{0}{It's zero!}{It's not zero!}
\pgfmathfloatifflags{\pgfmathresult}{1}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{2}{It's negative!}{It's not negative!}

% or, equivalently
\pgfmathfloatifflags{\pgfmathresult}{+}{It's positive!}{It's not positive!}
\pgfmathfloatifflags{\pgfmathresult}{-}{It's negative!}{It's not negative!}
```

\pgfmathfloattomacro{⟨*x*⟩}{⟨*flagsmacro*⟩}{⟨*mantissamacro*⟩}{⟨*exponentmacro*⟩}

Extracts the flags of a floating point number ⟨*x*⟩ to ⟨*flagsmacro*⟩, the mantissa to ⟨*mantissamacro*⟩ and the exponent to ⟨*exponentmacro*⟩.

\pgfmathfloattoregisters{⟨*x*⟩}{⟨*flagscount*⟩}{⟨*mantissadimen*⟩}{⟨*exponentcount*⟩}

Takes a floating point number ⟨*x*⟩ as input and writes flags to count register ⟨*flagscount*⟩, mantissa to dimen register ⟨*mantissadimen*⟩ and exponent to count register ⟨*exponentcount*⟩.

Please note that this method rounds the mantissa to TEX-precision.

\pgfmathfloattoregisterstok{⟨*x*⟩}{⟨*flagscount*⟩}{⟨*mantissatoks*⟩}{⟨*exponentcount*⟩}

A variant of \pgfmathfloattoregisters which writes the mantissa into a token register. It maintains the full input precision.

\pgfmathfloatgetflags{⟨*x*⟩}{⟨*flagscount*⟩}

Extracts the flags of ⟨*x*⟩ into the count register ⟨*flagscount*⟩.

\pgfmathfloatgetflagstomacro{⟨*x*⟩}{⟨*macro*⟩}

Extracts the flags of ⟨*x*⟩ into the macro ⟨*macro*⟩.

\pgfmathfloatgetmantissa{⟨*x*⟩}{⟨*mantissadimen*⟩}

Extracts the mantissa of ⟨*x*⟩ into the dimen register ⟨*mantissadimen*⟩.

\pgfmathfloatgetmantissatok{⟨*x*⟩}{⟨*mantissatoks*⟩}

Extracts the mantissa of ⟨*x*⟩ into the token register ⟨*mantissatoks*⟩.

\pgfmathfloatgetexponent{⟨*x*⟩}{⟨*exponentcount*⟩}

Extracts the exponent of ⟨*x*⟩ into the count register ⟨*exponentcount*⟩.

56.4.2 Symbolic Rounding Operations

Commands in this section constitute the basic level implementations of the rounding routines. They work symbolically, i.e. they operate on text, not on numbers and allow arbitrarily large numbers.

\pgfmathroundto{⟨*x*⟩}

Rounds a fixed point number to prescribed precision and writes the result to `\pgfmathresult`.

The desired precision can be configured with `/pgf/number format/precision`, see section 97. This section does also contain application examples.

Any trailing zeros after the period are discarded. The algorithm is purely text based and allows to deal with precisions beyond TEX's fixed point support.

As a side effect, the global boolean `\ifpgfmathfloatroundhasperiod` will be set to true if and only if the resulting mantissa has a period. Furthermore, `\ifpgfmathfloatroundmayneedrenormalize` will be set to true if and only if the rounding result's floating point representation would have a larger exponent than ⟨*x*⟩.

1
```
\pgfmathroundto{1}
\pgfmathresult
```

4.69
```
\pgfmathroundto{4.685}
\pgfmathresult
```

20000
```
\pgfmathroundto{19999.9996}
\pgfmathresult
```

\pgfmathroundtozerofill{⟨*x*⟩}

A variant of `\pgfmathroundto` which always uses a fixed number of digits behind the period. It fills missing digits with zeros.

1.00
```
\pgfmathroundtozerofill{1}
\pgfmathresult
```

4.69
```
\pgfmathroundto{4.685}
\pgfmathresult
```

20000.00
```
\pgfmathroundtozerofill{19999.9996}
\pgfmathresult
```

\pgfmathfloatround{⟨*x*⟩}

Rounds a normalized floating point number to a prescribed precision and writes the result to `\pgfmathresult`.

The desired precision can be configured with `/pgf/number format/precision`, see section 97.

This method employs `\pgfmathroundto` to round the mantissa and applies renormalization if necessary.

As a side effect, the global boolean `\ifpgfmathfloatroundhasperiod` will be set to true if and only if the resulting mantissa has a period.

5.26e1
```
\pgfmathfloatparsenumber{52.5864}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

1e1
```
\pgfmathfloatparsenumber{9.995}
\pgfmathfloatround{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

`\pgfmathfloatroundzerofill{`⟨*x*⟩`}`

A variant of `\pgfmathfloatround` produces always the same number of digits after the period (it includes zeros if necessary).

5.26e1
```
\pgfmathfloatparsenumber{52.5864}
\pgfmathfloatroundzerofill{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

1.00e1
```
\pgfmathfloatparsenumber{9.995}
\pgfmathfloatroundzerofill{\pgfmathresult}
\pgfmathfloattosci{\pgfmathresult}
\pgfmathresult
```

56.4.3 Math Operations Commands

This section describes some of the replacement commands in more detail.

Please note that these commands can be used even if the `fpu` as such has not been activated – it is sufficient to load the library.

`\pgfmathfloat`⟨*op*⟩

Methods of this form constitute the replacement operations where ⟨*op*⟩ can be any of the well-known math operations.

Thus, `\pgfmathfloatadd` is the counterpart for `\pgfmathadd` and so on. The semantics and number of arguments is the same, but all input and output arguments are *expected* to be floating point numbers.

`\pgfmathfloattoextentedprecision{`⟨*x*⟩`}`

Renormalizes ⟨*x*⟩ to extended precision mantissa, meaning $100 \leq m < 1000$ instead of $1 \leq m < 10$.

The "extended precision" means we have higher accuracy when we apply pgfmath operations to mantissas.

The input argument is expected to be a normalized floating point number; the output argument is a non-normalized floating point number (well, normalized to extended precision).

The operation is supposed to be very fast.

`\pgfmathfloatsetextprecision{`⟨*shift*⟩`}`

Sets the precision used inside of `\pgfmathfloattoextentedprecision` to ⟨*shift*⟩.

The different choices are

0	normalization to	0	$\leq m < 1$	(disable extended precision)
1	normalization to	10	$\leq m < 100$	
2	normalization to	100	$\leq m < 1000$	(default of `\pgfmathfloattoextentedprecision`)
3	normalization to	1000	$\leq m < 10000$	

`\pgfmathfloatlessthan{`⟨*x*⟩`}{`⟨*y*⟩`}`

Defines `\pgfmathresult` as 1.0 if $\langle x \rangle < \langle y \rangle$, but 0.0 otherwise. It also sets the global TEX-boolean `\pgfmathfloatcomparison` accordingly. The arguments $\langle x \rangle$ and $\langle y \rangle$ are expected to be numbers which have already been processed by `\pgfmathfloatparsenumber`. Arithmetic is carried out using TEX-registers for exponent- and mantissa comparison.

`\pgfmathfloatmultiplyfixed{`$\langle float \rangle$`}{`$\langle fixed \rangle$`}`

Defines `\pgfmathresult` to be $\langle float \rangle \cdot \langle fixed \rangle$ where $\langle float \rangle$ is a floating point number and $\langle fixed \rangle$ is a fixed point number. The computation is performed in floating point arithmetic, that means we compute $m \cdot \langle fixed \rangle$ and renormalize the result where m is the mantissa of $\langle float \rangle$.

This operation renormalizes $\langle float \rangle$ with `\pgfmathfloattoextentedprecision` before the operation, that means it is intended for relatively small arguments of $\langle fixed \rangle$. The result is a floating point number.

`\pgfmathfloatifapproxequalrel{`$\langle a \rangle$`}{`$\langle b \rangle$`}{`$\langle true\text{-}code \rangle$`}{`$\langle false\text{-}code \rangle$`}`

Computes the relative error between $\langle a \rangle$ and $\langle b \rangle$ (assuming $\langle b \rangle \neq 0$) and invokes $\langle true\text{-}code \rangle$ if the relative error is below `/pgf/fpu/rel thresh` and $\langle false\text{-}code \rangle$ if that is not the case.

The input arguments will be parsed with `\pgfmathfloatparsenumber`.

`/pgf/fpu/rel thresh=`{$\langle number \rangle$} (no default, initially `1e-4`)

A threshold used by `\pgfmathfloatifapproxequalrel` to decide whether numbers are approximately equal.

`\pgfmathfloatshift{`$\langle x \rangle$`}{`$\langle num \rangle$`}`

Defines `\pgfmathresult` to be $\langle x \rangle \cdot 10^{\langle num \rangle}$. The operation is an arithmetic shift base ten and modifies only the exponent of $\langle x \rangle$. The argument $\langle num \rangle$ is expected to be a (positive or negative) integer.

`\pgfmathfloatabserror{`$\langle x \rangle$`}{`$\langle y \rangle$`}`

Defines `\pgfmathresult` to be the absolute error between two floating point numbers x and y, $|x - y|$ and returns the result as floating point number.

`\pgfmathfloatrelerror{`$\langle x \rangle$`}{`$\langle y \rangle$`}`

Defines `\pgfmathresult` to be the relative error between two floating point numbers x and y, $|x-y|/|y|$ and returns the result as floating point number.

`\pgfmathfloatint{`$\langle x \rangle$`}`

Returns the integer part of the floating point number $\langle x \rangle$, by truncating any digits after the period. This methods truncates the absolute value $|x|$ to the next smaller integer and restores the original sign afterwards.

The result is returned as floating point number as well.

See also `\pgfmathfloattoint` which returns the number in integer format.

`\pgfmathlog{`$\langle x \rangle$`}`

Defines `\pgfmathresult` to be the natural logarithm of $\langle x \rangle$, $\ln(\langle x \rangle)$. This method is logically the same as `\pgfmathln`, but it applies floating point arithmetic to read number $\langle x \rangle$ and employs the logarithm identity

$$\ln(m \cdot 10^e) = \ln(m) + e \cdot \ln(10)$$

to get the result. The factor $\ln(10)$ is a constant, so only $\ln(m)$ with $1 \leq m < 10$ needs to be computed. This is done using standard pgf math operations.

Please note that $\langle x \rangle$ needs to be a number, expression parsing is not possible here.

If $\langle x \rangle$ is *not* a bounded positive real number (for example $\langle x \rangle \leq 0$), `\pgfmathresult` will be *empty*, no error message will be generated.

-15.7452

```
\usetikzlibrary {fpu}
\pgfmathlog{1.452e-7}
\pgfmathresult
```

20.28096

```
\usetikzlibrary {fpu}
\pgfmathlog{6.426e+8}
\pgfmathresult
```

56.4.4 Accessing the Original Math Routines for Programmers

As soon as the library is loaded, every private math routine will be copied to a new name. This allows library and package authors to access the TEX-register based math routines even if the FPU is activated. And, of course, it allows the FPU as such to perform its own mantissa computations.

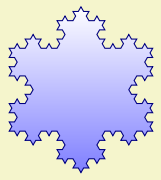
The private implementations of PGF math commands, which are of the form `\pgfmath`⟨*name*⟩`@`, will be available as`\pgfmath@basic@`⟨*name*⟩`@` as soon as the library is loaded.

57 Lindenmayer System Drawing Library

57.1 Overview

Lindenmayer systems (also commonly known as "L-systems"), were originally developed by Aristid Lindenmayer as a theory of algae growth patterns and then subsequently used to model branching patterns in plants and produce fractal patterns. Typically, an L-system consists of a set of symbols, each of which is associated with some graphical action (such as "turn left" or "move forward") and a set of rules ("production" or "rewrite" rules). Given a string of symbols, the rewrite rules are applied several times and when the resulting string is processed the action associated with each symbol is executed.

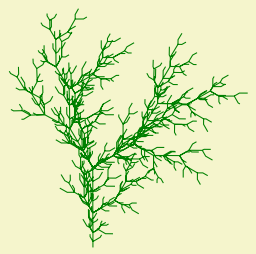
In PGF, L-systems can be used to create simple 2-dimensional fractal patterns…



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
\pgfdeclarelindenmayersystem{Koch curve}{
  \rule{F -> F-F++F-F}
}

\shadedraw [top color=white, bottom color=blue!50, draw=blue!50!black]
  [l-system={Koch curve, step=2pt, angle=60, axiom=F++F++F, order=3}]
  lindenmayer system -- cycle;
\end{tikzpicture}
```

…and "plant like" patterns…



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
\draw [green!50!black, rotate=90]
  [l-system={rule set={F -> FF-[-F+F]+[+F-F]}, axiom=F, order=4, step=2pt,
   randomize step percent=25, angle=30, randomize angle percent=5}]
  lindenmayer system;
\end{tikzpicture}
```

…but it is important to bear in mind that even moderately complex L-systems can exceed the available memory of TEX, and can be very slow. If possible, you are advised to increase the main memory and save stack to their maximum possible values for your particular TEX distribution. However, even by doing this you may find you still run out of memory quite quickly.

For an excellent introduction to L-systems (containing some "really cool" pictures – many of which are sadly not possible in PGF) see *The Algorithmic Beauty of Plants* by Przemyslaw Prusinkiewicz and Aristid Lindenmayer (which is freely available via the internet).

TikZ Library `lindenmayersystems`

```
\usepgflibrary{lindenmayersystems}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[lindenmayersystems]  % ConTeXt when using pure PGF
\usetikzlibrary{lindenmayersystems} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[lindenmayersystems] % ConTeXt when using TikZ
```

This PGF-library provides basic commands for defining and using simple L-systems. The TikZ-library provides, furthermore, a front end for using L-systems in TikZ.

57.1.1 Declaring L-systems

Before an L-system can be used, it must be declared using the following command:

`\pgfdeclarelindenmayersystem{`⟨*name*⟩`}{`⟨*specification*⟩`}`

This command declares a Lindenmayer system called ⟨*name*⟩. The ⟨*specification*⟩ argument contains a description of the L-system's symbols and rules. Two commands `\symbol` and `\rule` are only defined when the ⟨*specification*⟩ argument is executed.

`\symbol{`⟨*name*⟩`}{`⟨*code*⟩`}`

This defines a symbol called ⟨*name*⟩ for a specific L-system, and associates it with ⟨*code*⟩.

A symbol should consist of a single alpha-numeric character (i.e., `A-Z`, `a-z` or `0-9`). The symbols `F`, `f`, `+`, `-`, `[` and `]` are available by default so do not need to be defined for each L-system. However, if you are feeling adventurous, they can be redefined for specific L-systems if required. The L-system treats the default symbols as follows (the commands they execute are described below):

- `F` move forward a certain distance, drawing a line. Uses `\pgflsystemdrawforward`.
- `f` move forward a certain distance, without drawing a line. Uses `\pgflsystemmoveforward`.
- `+` turn left by some angle. Uses `\pgflsystemturnleft`.
- `-` turn right by some angle. Uses `\pgflsystemturnright`.
- `[` save the current state (i.e., the position and direction). Uses `\pgflsystemsavestate`.
- `]` restore the last saved state. Uses `\pgflsystemrestorestate`.

The symbols `[` and `]` act like a stack: `[` pushes the state of the L-system on to the stack, and `]` pops a state off the stack.

When ⟨*code*⟩ is executed, the transformation matrix is set up so that the origin is at the current position and the positive x-axis "points forward", so `\pgfpathlineto{\pgfpoint{1cm}{0cm}}` draws a line 1cm forward.

The following keys can alter the production of an L-system. However, they do not store values in themselves.

`/pgf/lindenmayer system/step`=⟨*length*⟩ (no default, initially `5pt`)

How far the L-system moves forward if required. This key sets the TEX dimension `\pgflsystemstep`.

`/pgf/lindenmayer system/randomize step percent`=⟨*percentage*⟩ (no default, initially `0`)

If the step is to be randomized, this key specifies by how much. The value is stored in the TEX macro `\pgflsystemrandomizesteppercent`.

`/pgf/lindenmayer system/left angle`=⟨*angle*⟩ (no default, initially `90`)

This key sets the angle through which the L-system turns when it turns left. The value is stored in the TEX macro `\pgflsystemrleftangle`.

`/pgf/lindenmayer system/right angle`=⟨*angle*⟩ (no default, initially `90`)

This key sets the angle through which the L-system turns when it turns right. The value is stored in the TEX macro `\pgflsystemrrightangle`.

`/pgf/lindenmayer system/randomize angle percent`=⟨*percentage*⟩ (no default, initially `0`)

If the angles are to be randomized, this key specifies by how much. The value is stored in the TEX macro `\pgflsystemrandomizeanglepercent`.

For speed and convenience, when the code for a symbol is executed, the following commands are available.

`\pgflsystemcurrentstep`

The current "step" of the L-system (i.e., how far the system will move forward if required). This is initially set to the value in the TEX-dimensions `\pgflsystemstep`, but the actual value may be changed if `\pgflsystemrandomizestep` is used (see below).

`\pgflsystemcurrentleftangle`

The angle the L-system will turn when it turns left. The value stored in this macro may be changed if `\pgflsystemrandomizeleftangle` is used.

`\pgflsystemcurrentrightangle`

The angle the L-system will turn when it turns right. The value stored in this macro may be changed if `\pgflsystemrandomizerightangle` is used.

The following commands may be useful if you wish to define your own symbols.

`\pgflsystemrandomizestep`

Randomizes the value in `\pgflsystemcurrentstep` according to the current value of the key `randomize step percent`.

\pgflsystemrandomizeleftangle

Randomizes the value in \pgflsystemcurrentleftangle according to the value of the randomize angle percent key.

\pgflsystemrandomizerightangle

Randomizes the value in \pgflsystemcurrentrightangle according to the value of the randomize angle key.

\pgflsystemdrawforward

Move forward in the current direction, by \pgflsystemcurrentstep, drawing a line in the process. This macro calls \pgflsystemrandomizestep. Internally, PGF simply shifts the transformation matrix in the positive direction of the current (transformed) x-axis by \pgflsystemstep and then executes a line-to to the (newly transformed) origin.

\pgflsystemmoveforward

Move forward in the current direction, by \pgflsystemcurrentstep, without drawing a line. This macro calls \pgflsystemrandomizestep. PGF executes a transformation as above, but executes a move-to to the (newly transformed) origin.

\pgflsystemturnleft

Turn left by \pgflsystemcurrentleftangle. Internally, PGF simply rotates the transformation matrix. This macro calls \pgflsystemrandomizeleftangle.

\pgflsystemturnright

Turn right by \pgflsystemcurrentrightangle. Internally, PGF simply rotates the transformation matrix. This macro calls \pgflsystemrandomizerightangle.

\pgflsystemsavestate

Save the current position and orientation. Internally, PGF simply starts a new TEX-group.

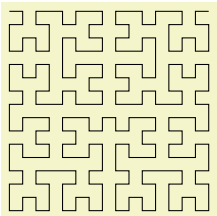
\pgflsystemrestorestate

Restore the last saved position and orientation. Internally, PGF closes a TEX-group, restoring the transformation matrix of the outer scope, and a move-to command is executed to the (transformed) origin.

\rule{⟨*head*⟩->⟨*body*⟩}

Declare a rule. ⟨*head*⟩ should consist of a single symbol, which need not have been declared using \symbol or exist as a default symbol (in fact, the more interesting L-systems depend on using symbols with no corresponding code, to control the "growth" of the system). ⟨*body*⟩ consists of a string of symbols, which again need not necessarily have any code associated with them.

As an example, the following shows an L-system that uses some of these commands. This example illustrates the point that some symbols, in this case A and B, do not have to have code associated with them. They simply control the growth of the system.



```
\usetikzlibrary {lindenmayersystems}
\pgfdeclarelindenmayersystem{Hilbert curve}{
  \symbol{X}{\pgflsystemdrawforward}
  \symbol{+}{\pgflsystemturnright} % Explicitly define + and - symbols.
  \symbol{-}{\pgflsystemturnleft}
  \rule{A -> +BX-AXA-XB+}
  \rule{B -> -AX+BXB+XA-}
}
\tikz\draw[lindenmayer system={Hilbert curve, axiom=A, order=4, angle=90}]
  lindenmayer system;
```

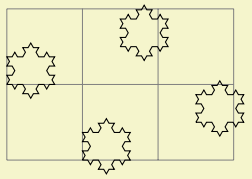
57.2 Using Lindenmayer Systems

57.2.1 Using L-Systems in PGF

The following command is used to run an L-system in PGF:

\pgflindenmayersystem{⟨*name*⟩}{⟨*axiom*⟩}{⟨*order*⟩}

Runs the L-system called ⟨*name*⟩ using the input string ⟨*axiom*⟩ for ⟨*order*⟩ iterations. In general, prior to calling this command, the transformation matrix should be set appropriately for shifting and rotating, and a move-to to the (transformed) origin should be executed. This origin will be where the L-system starts. In addition, the relevant keys should be set appropriately.



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \pgfset{lindenmayer system/.cd, angle=60, step=2pt}
  \foreach \x/\y in {0cm/1cm, 1.5cm/1.5cm, 2.5cm/0.5cm, 1cm/0cm}{
    \pgftransformshift{\pgfqpoint{\x}{\y}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgflindenmayersystem{Koch curve}{F++F++F}{2}
    \pgfusepath{stroke}
  }
\end{tikzpicture}
```

Note that it is perfectly feasible for an L-system to define special symbols which perform the move-to and use-path operations.

57.2.2 Using L-Systems in Ti*k*Z

In Ti*k*Z, an L-system is created using a path operation. However, Ti*k*Z is more flexible regarding the positioning of the L-system and also provides keys to create L-systems "on-line".

\path … lindenmayer system [⟨*keys*⟩] …;

This will run an L-system according to the parameters specified in ⟨*keys*⟩ (which can also contain normal keys such as `draw` or `thin`). The syntax is flexible regarding the L-system parameters and the following all do the same thing:

```
\draw lindenmayer system [lindenmayer system={Hilbert curve, axiom=4, order=3}];
```

```
\draw [lindenmayer system={Hilbert curve, axiom=4, order=3}] lindenmayer system;
```

```
\tikzset{lindenmayer system={Hilbert curve, axiom=4, order=3}}
\draw lindenmayer system;
```

\path … l-system [⟨*keys*⟩] …;

A more compact version of the `lindenmayer system` path command.

This library adds some additional keys for specifying L-systems. These keys only work in Ti*k*Z and all have the same path, namely, `/pgf/lindenmayer system`, but the following keys are provided for convenience, so that you do not have to keep repeating this path:

/pgf/lindenmayer system={⟨*keys*⟩} (style, no default)

alias `/tikz/lindenmayer system`

This key changes the key path to `/pgf/lindenmayer systems` and executes ⟨*keys*⟩.

/pgf/l-system={⟨*keys*⟩} (style, no default)

alias `/tikz/l-system`

A more compact version of the previous key.

/pgf/lindenmayer system/name={⟨*name*⟩} (no default)

Sets the name for the L-system.

/pgf/lindenmayer system/axiom={⟨*string*⟩} (no default)

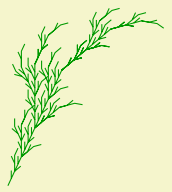
Sets the axiom (or input string) for the L-system.

/pgf/lindenmayer system/order={⟨*integer*⟩} (no default)

Sets the number of iterations the L-system will perform.

/pgf/lindenmayer system/rule set={⟨*list*⟩} (no default)

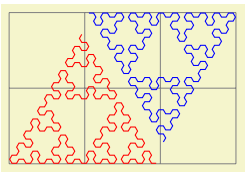
This key allows an (anonymous) L-system to be declared "on-line". There is, however, a restriction that only the default symbols can be used for drawing (empty symbols can still be used to control the growth of the system). The rules in ⟨*list*⟩ should be separated by commas.



```
\usetikzlibrary {lindenmayersystems}
\tikz[rotate=65]\draw [green!60!black] l-system
  [l-system={rule set={F -> F[+F]F[-F]}, axiom=F, order=4, angle=25,step=3pt}];
```

/pgf/lindenmayer system/anchor=⟨*anchor*⟩ (no default)

Be default, when this key is not used, the L-system will start from the last specified coordinate. By using this key, the L-system will be placed inside a special (rectangle) node which can be positioned using ⟨*anchor*⟩.



```
\usetikzlibrary {lindenmayersystems}
\begin{tikzpicture}[l-system={step=1.75pt, order=5, angle=60}]
  \pgfdeclarelindenmayersystem{Sierpinski triangle}{
    \symbol{X}{\pgflsystemdrawforward}
    \symbol{Y}{\pgflsystemdrawforward}
    \rule{X -> Y-X-Y}
    \rule{Y -> X+Y+X}
  }
  \draw [help lines] grid (3,2);
  \draw [red] (0,0) l-system
    [l-system={Sierpinski triangle, axiom=+++X, anchor=south west}];
  \draw [blue] (3,2) l-system
    [l-system={Sierpinski triangle, axiom=X, anchor=north east}];
\end{tikzpicture}
```

58 Math Library

Ti*k*Z Library math

```
\usetikzlibrary{math} % LaTeX and plain TeX
\usetikzlibrary[math] % ConTeXt
```

This library defines a simple mathematical language to define simple functions and perform sequences of basic mathematical operations.

58.1 Overview

PGF and Ti*k*Z both use the PGF mathematical engine which provides many commands for parsing expressions. Unfortunately the PGF math engine is somewhat cumbersome for long sequences of mathematical operations, particularly when assigning values to multiple variables. The Ti*k*Z `calc` library provides some additional "convenience" operations for doing calculations (particularly with coordinates), but this can only be used inside Ti*k*Z path commands.

This `math` library provides a means to perform sequences of mathematical operations in a more 'user friendly' manner than the PGF math engine. In addition, the coordinate calculations of the `calc` library can be accessed (provided it is loaded). However as the `math` library uses the PGF math engine – which uses pure TeX to perform all its calculations – it is subject to the same speed and accuracy limitations. It is worth bearing this in mind, before trying to implement algorithms requiring intensive and highly accurate computation. You can, of course use the `fp` or the `fpu` libraries to increase the accuracy (but not necessarily the speed) of computations.

For most purposes, the features provided by this library are accessed using the following command:

`\tikzmath{`⟨*statements*⟩`}`

This command process a series of ⟨*statements*⟩ which can represent assignments, function definitions, conditional evaluation, and iterations. It provides, in effect, a miniature mathematical language to perform basic mathematical operations. Perhaps the most important thing to remember is that *every statement should end with a semi-colon.* This is likely to be the most common reason why the `\tikzmath` command fails.

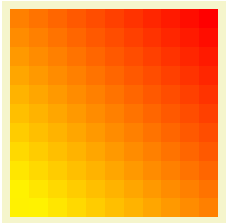
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765,

```
\usetikzlibrary {math}
\tikzmath{
  % Adapted from http://www.cs.northwestern.edu/academics/courses/110/html/fib_rec.html
  function fibonacci(\n) {
    if \n == 0 then {
      return 0;
    } else {
      return fibonacci2(\n, 0, 1);
    };
  };
  function fibonacci2(\n, \p, \q) {
    if \n == 1 then {
      return \q;
    } else {
      return fibonacci2(\n-1, \q, \p+\q);
    };
  };
  int \f, \i;
  for \i in {0,1,...,20} {
    \f = fibonacci(\i);
    print {\f, };
  };
}
```

In addition to this command the following key is provided:

`/tikz/evaluate=`⟨*statements*⟩ (no default)

This key simply executes `\tikzmath{`⟨*statements*⟩`}`.



```
\usetikzlibrary {math}
\tikz[x=0.25cm,y=0.25cm,
  evaluate={
    int \i, \j;
    for \i in {0,...,10} {
      for \j in {0,...,10} {
        \a{\i,\j} = (\i+\j)*5;
      };
    };
  }
]
\foreach \i in {0,...,10}
  \foreach \j in {0,...,10}
    \fill [red!\a{\i,\j}!yellow]  (\i,\j) rectangle ++(1, 1);
```

The following sections describe the miniature language that this library provides and can be used in the `\tikzmath` command and the `evaluate` key. The language consists only of simple keywords and expressions but the mini-parser allows you to format code in a reasonably versatile way (much like the `tikz` parser) except that *all the keywords must be followed by at least one space*. This is the second most important thing to remember (after remembering to insert semi-colons at the end of every statement).

58.2 Assignment

In the simplest case, you will want to evaluate an expression and assign it to a macro, or a TEX count or dimension register. In this case, use of the `math` library is straightforward:

26.0, 2.0, 11, 225.0pt

```
\usetikzlibrary {math}
\newcount\mycount
\newdimen\mydimen
\tikzmath{
  \a = 4*5+6;
  \b = sin(30)*4;
  \mycount = log10(2048) / log10(2);
  \mydimen = 15^2;
}
\a, \b, \the\mycount, \the\mydimen
```

In addition, TEX-macros (*not* TEX registers) can be suffixed with an index, similar to indices in mathematical notation, for example, x_1, x_2, x_3:

7.0, 70.0, 700.0

```
\usetikzlibrary {math}
\tikzmath{
  \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

The index does not have to be a number. By using braces `{}`, more sophisticated indices can be created:

The speed of sound in air is 340 m/s. The speed of sound in steel is 6100 m/s.

```
\usetikzlibrary {math}
\tikzmath{
  \c{air} = 340; \c{water} = 1435; \c{steel} = 6100;
}
\foreach \medium in {air,steel}{The speed of sound in \medium\ is \c{\medium} m/s. }
```

You should not, however, try to mix indexed and non-indexed variables. Once an assignment is made using an index, the `math` library expects all instances of the variable on the right hand side of an assignment to be followed by an index. This effect is reversed if you subsequently make an assignment to the variable without an index: the `math` library (or to be precise the PGF math-engine) will then ignore any index following the variable on the right hand side of an assignment.

In some cases, you may wish to assign a value or expression to a variable without evaluating it with the PGF math-engine. In this case, you can use the following keyword:

`let` ⟨*variable*⟩ `=` ⟨*expression*⟩`;`

This keyword assigns ⟨*expression*⟩ to ⟨*variable*⟩ without evaluation. The ⟨*expression*⟩ is however fully expanded using `\edef`. Any spaces preceding ⟨*expression*⟩ are removed, but any trailing spaces (before the semi-colon) are included.

(5*4)+1, “blue”

```
\usetikzlibrary {math}
\tikzmath{
  let \x = (5*4)+1;
  let \c1 = blue;
}
\x, ``\c1''
```

58.3 Integers, “Real” Numbers, and Coordinates

By default, assignments are made by evaluating expressions using the PGF math-engine and results are usually returned as number with a decimal point (unless you are assigning to a count register or use the `int` function). As this is not always desirable, the `math` library allows variables – which *must* be TEX macros – to be ‘declared’ as being a particular ‘type’. The library recognizes three types: integers (numbers without a decimal point), real numbers (numbers with a decimal point[12]), and coordinates.

To declare a variable as being one of the three types, you can use the keywords shown below. It is important to remember that by telling the `math` library you want it to do a particular assignment for a variable, it will also do the same assignment when the variable is indexed.

7, 70, 700

```
\usetikzlibrary {math}
\tikzmath{
  integer \x;
  \x1 = 3+4; \x2 = 30+40; \x3 = 300+400;
}
\x1, \x2, \x3
```

`integer` ⟨*variable*⟩`,` ⟨*additional variables*⟩`;`

The `integer` keyword indicates that assignments to the ⟨*variable*⟩ or the comma separated list of ⟨*additional variables*⟩ should be truncated (not rounded) to integers. The variables should be ordinary macros – *not* TEX registers. In addition the variables should *not* be indexed.

$x = 26$, $y = 2$, $z = 9$

```
\usetikzlibrary {math}
\tikzmath{
   integer \x, \y, \z;
   \x = 4*5+6;
   \y = sin(30)*4;
   \z = log10(512) / log10(2);
   print {$x=\x$, $y=\y$, $z=\z$};
}
```

`int` ⟨*variable*⟩`,` ⟨*additional variables*⟩`;`

Short version of the `integer` keyword.

Having declared a variable as an integer, the `math` library will continue to assign only integers to that variable within the current TEX scope. If you wish to assign non-integer (i.e., *real*) numbers to the same variable, the following keyword can be used.

`real` ⟨*variable*⟩`,` ⟨*additional variables*⟩`;`

The `real` keyword ensures that assignments ⟨*variable*⟩ (and ⟨*additional variables*⟩) will not be truncated to integers.

In order to take advantage of `math` library interface to the `calc` library you must indicate that a variable is to be assigned coordinates, using the following keyword.

`coordinate` ⟨*variable*⟩`,` ⟨*additional variables*⟩`;`

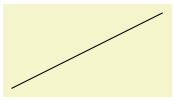
This keyword enables Ti*k*Z-style coordinates such as `(2cm,3pt)` or `(my node.east)` to be parsed and assigned to ⟨*variable*⟩ in the form x, y, which can then be used in a `tikzpicture`:

[12] Strictly speaking, due to the finite range and precision of TEX numerical capabilities, the term “real” is not correct.



```
\usetikzlibrary {math}
\tikzmath{
   coordinate \c;
   \c = (45:10pt);
}
\tikz\draw (0,0) -- (\c);
```

If the Ti*k*Z `calc` library is loaded, coordinate calculations can be performed; the coordinate expression does not have to be surrounded by `($…$)`.



```
\usetikzlibrary {math,calc}
\tikzmath{
   coordinate \c, \d;
   \c = (-1,2)+(1,-1);
   \d = (4,1)-(2,-1);
}
\tikz\draw (\c) -- (\d);
```

In addition to assigning the x and y coordinates to ⟨*variable*⟩ (possibly with an optional index), two further variables are defined. The first takes the name of ⟨*variable*⟩ (e.g., `\c`) suffixed with `x` (i.e., `\cx`) and is assigned the x coordinate of `\c`. The second takes the name of ⟨*variable*⟩ suffixed with `y` (i.e., `\cy`) and is assigned the y coordinate of `\c`.



```
\usetikzlibrary {math}
\tikzmath{
   coordinate \c;
   \c1 = (30:20pt);
   \c2 = (210:20pt);
}
\tikz\draw (\c1,\cy1) -- (\c2,\cy1) -- (\c2,\cy2) -- (\c1,\cy2);
```

58.4 Repeating Things

for ⟨*variable*⟩ `in {`⟨*list*⟩ `}{`⟨*expressions*⟩`};`

This is a "trimmed down" version of the `\foreach` command available as part of PGF and Ti*k*Z, but cannot currently be used outside of the `\tikzmath` command. It is important to note the following:

- Every value in ⟨*list*⟩ is evaluated using the PGF mathematical engine. However, if an item in ⟨*list*⟩ contains a comma, it *must* be surrounded by braces, for example, `{mod(5, 2)}`.

$x = 8, v = 256$

```
\usetikzlibrary {math}
\tikzmath{
  int \x, \v;
  \v=1;
  for \x in {1,...,{random(3,10)}} {
     \v=\v*2;
  };
  print {$x=\x, v=\v$};
}
```

- Because each item is evaluated, you cannot use Ti*k*Z coordinates in ⟨*list*⟩.
- Only single variable assignment is supported.
- The "dots notation" (e.g., `1,2,...,9`) can be used in ⟨*list*⟩, but is not as sophisticated as the PGF `\foreach` command. In particular, contextual replacement is not possible.
- Assignments that occur in the loop body *are not scoped*. They last beyond the body of each iteration and the end of the `for` statement. This includes the values assigned to the ⟨*variable*⟩.

$x_1 = 5, x_2 = 50, y = 2250$

```
\usetikzlibrary {math}
\tikzmath{
  int \x, \y;
  \y = 0;
  for \x1 in {1,...,5} {
    for \x2 in {10,20,...,50} {
      \y = \y+\x1*\x2;
    };
  };
}
$x_1=\x1, x_2=\x2, y=\y$
```

58.5 Branching Statements

Sometimes you may wish to execute different statements depending on the value of an expression. In this case the following keyword can be used:

`if` ⟨*condition*⟩ `then {`⟨*if-non-zero-statements*⟩`};`

This keyword executes ⟨*if-non-zero-statements*⟩ if the expression in ⟨*condition*⟩ evaluates to any value other than zero.

`if` ⟨*condition*⟩ `then {`⟨*if-non-zero-statements*⟩`} else {`⟨*if-zero-statements*⟩`};`

This keyword executes ⟨*if-non-zero-statements*⟩ if the expression in ⟨*condition*⟩ evaluates to any value other than zero and the ⟨*if-zero-statements*⟩ are executed if the expression in ⟨*condition*⟩ evaluates to zero.



```
\usetikzlibrary {math}
  \begin{tikzpicture}
  \tikzmath{
    int \k;
    for \k in {0,10,...,350} {
      if \k>260 then { let \c = orange; } else {
        if \k>170 then { let \c = blue; } else {
          if \k>80 then { let \c = red; } else {
            let \c = green; }; }; };
      {
        \path [fill=\c!50, draw=\c] (\k:0.5cm) -- (\k:1cm) --
          (\k+5:1cm) -- (\k+5:0.5cm) -- cycle;
      };
    };
  }
  \end{tikzpicture}
```

58.6 Declaring Functions

You can add functions by using the following keywords:

`function` ⟨*name*⟩`(`⟨*arguments*⟩`) {` ⟨*definition*⟩ `};`

This keyword works much like the `declare function` provided by the PGF math-engine. The function ⟨*name*⟩ can be any name that is not already a function name in the current scope. The list of ⟨*arguments*⟩ are comma separated TEX macros such as `\x`, or `\y` (it is not possible to declare functions that take variable numbers of arguments). If the function takes no arguments then the parentheses need not be used. It is very important to note that the arrays that the PGF math engine supports *cannot currently be passed as arguments to functions.*

The function ⟨*definition*⟩ should be a sequence of statements that can be parsed by the `\tikzmath` command and should use the commands specified in the ⟨*arguments*⟩. The `return` keyword (described below) should be used to indicate the value returned by the function. Although ⟨*definition*⟩ can take any statements accepted by `\tikzmath`, it is not advisable try to define functions inside other functions.

$2 \times 33 = 66$

```
\usetikzlibrary {math}
\tikzmath{
  function product(\x,\y) {
    return \x*\y;
  };
  int \i, \j, \k;
  \i = random(1,10);
  \j = random(20, 40);
  \k = product(\i, \j);
  print { $\i\times \j = \k$ };
}
```

`return` ⟨*expression*⟩`;`

This keyword should be used as the last executed statement in a function definition to indicate the value that should be returned.

58.7 Executing Code Outside the Parser

Sometimes you may wish to do "something" outside the parser, perhaps display some intermediate result or execute some code. In this case you have two options. Firstly, the following keyword can be used:

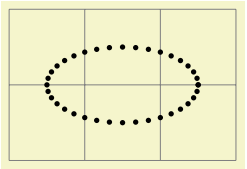
`print {`⟨*code*⟩`};`

Execute ⟨*code*⟩ immediately. This is intended as convenience keyword for displaying information in a document (analogous to the `print` command in real programming languages). The ⟨*code*⟩ is executed inside a TEX group.

$3^0 = 1,\ 3^1 = 3,\ 3^2 = 9,\ 3^3 = 27,\ 3^4 = 81,\ 3^5 = 243,\ 3^6 = 729,$

```
\usetikzlibrary {math}
\tikzmath{
  int \x, \y, \z;
  \x = random(2, 5);
  for \y in {0,...,6} {
    \z = \x^\y;
    print {$\x^\y=\z$, };
  };
}
```

Secondly, if a statement begins with a brace `{`, then everything up to the closing brace `}` is collected and executed (the closing brace *must* be followed by a semi-colon). Like the `print` keyword, the contents of the braces is executed inside a TEX group. Unlike the `print` keyword, the brace notation can be used in functions so that `tikz` path commands can be safely executed inside a `tikzpicture`.



```
\usetikzlibrary {math,calc}
\begin{tikzpicture}
\draw [help lines] grid (3,2);
\tikzmath{
  coordinate \c;
  for \x in {0,10,...,360} {
    \c = (1.5cm, 1cm) + (\x:1cm and 0.5cm);
    { \fill (\c) circle [radius=1pt]; };
  };
}
\end{tikzpicture}
```

59 Matrix Library

TikZ Library matrix

```
\usetikzlibrary{matrix} % LaTeX and plain TeX
\usetikzlibrary[matrix] % ConTeXt
```

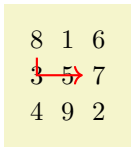
This library package defines additional styles and options for creating matrices. The basic matrix styles and options can be found in Section 20.

59.1 Matrices of Nodes

A *matrix of nodes* is a TikZ matrix in which each cell contains a node. In this case it is bothersome having to write `\node{` at the beginning of each cell and `};` at the end of each cell. The following key simplifies typesetting such matrices.

`/tikz/matrix of nodes` (no value)

Conceptually, this key adds `\node{` at the beginning and `};` at the end of each cell and sets the `anchor` of the node to `base`. Furthermore, it adds the option `name` option to each node, where the name is set to ⟨*matrix name*⟩`-`⟨*row number*⟩`-`⟨*column number*⟩. For example, if the matrix has the name `my matrix`, then the node in the upper left cell will get the name `my matrix-1-1`.




```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix (magic) [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };

  \draw[thick,red,->] (magic-1-1) |- (magic-2-3);
\end{tikzpicture}
```

You may wish to add options to certain nodes in the matrix. This can be achieved in three ways.

1. You can modify, say, the `row 2 column 3` style to pass special options to this particular cell.




```
\usetikzlibrary {matrix}
\begin{tikzpicture}[row 2 column 3/.style=red]
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & 7 \\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

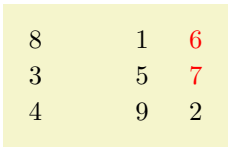
2. At the beginning of a cell, you can use a special syntax. If a cell starts with a vertical bar, then everything between this bar and the next bar is passed on to the `node` command.




```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1 &         6 \\
    3 & 5 & |[red]| 7 \\
    4 & 9 &         2 \\
  };
\end{tikzpicture}
```

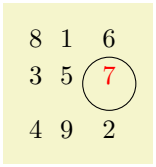
You can also use an option like `|[red] (seven)|` to give a different name to the node.

Note that the `&` character also takes an optional argument, which is an extra column skip.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 &[1cm] 1 &[3mm] |[red]| 6 \\
    3 &      5 &      |[red]| 7 \\
    4 &      9 &              2 \\
  };
\end{tikzpicture}
```

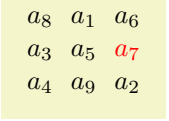
3. If your cell starts with a `\path` command or any command that expands to `\path`, which includes `\draw`, `\node`, `\fill` and others, the `\node{` startup code and the `};` code are suppressed. This means that for this particular cell you can provide totally different contents.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of nodes]
  {
    8 & 1 & 6 \\
    3 & 5 & \node[red]{7}; \draw(0,0) circle[radius=10pt];\\
    4 & 9 & 2 \\
  };
\end{tikzpicture}
```

`/tikz/matrix of math nodes` (no value)

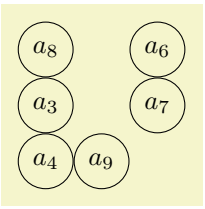
This style is almost the same as the previous style, only `$` is added at the beginning and at the end of each node, so math mode will be switched on in all nodes.



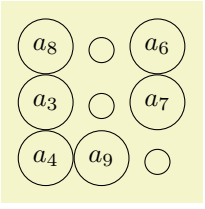
```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes]
  {
    a_8 & a_1 &         a_6 \\
    a_3 & a_5 & |[red]| a_7 \\
    a_4 & a_9 &         a_2 \\
  };
\end{tikzpicture}
```

`/tikz/nodes in empty cells=`⟨*true or false*⟩ (default `true`)

When set to `true`, a node (with empty contents) is put in empty cells. Normally, empty cells are just, well, empty. The style can be used together with both a `matrix of nodes` and a `matrix of math nodes`.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,nodes={circle,draw}]
  {
    a_8 &     & a_6 \\
    a_3 &     & a_7 \\
    a_4 & a_9 &     \\
  };
\end{tikzpicture}
```



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,nodes={circle,draw},nodes in empty cells]
  {
    a_8 &     & a_6 \\
    a_3 &     & a_7 \\
    a_4 & a_9 &     \\
  };
\end{tikzpicture}
```

59.2 End-of-Lines and End-of-Row Characters in Matrices of Nodes

Special care must be taken about the usage of the `\\` command inside a matrix of nodes. The reason is that this character is overloaded in TEX: On the one hand, it is used to denote the end of a line in normal text;

on the other hand it is used to denote the end of a row in a matrix. Now, if a matrix contains node which in turn may have multiple lines, it is unclear which meaning of \\ should be used.

This problem arises only when you use the `text width` option of nodes. Suppose you write a line like

```
\matrix [text width=5cm,matrix of nodes]
{
  first row & upper line \\ lower line \\
  second row & hmm \\
};
```

This leaves TEX trying to riddle out how many rows this matrix should have. Do you want two rows with the upper right cell containing a two-line text. Or did you mean a three row matrix with the second row having only one cell?

Since TEX is not clairvoyant, the following rules are used:

1. Inside a matrix, the \\ command, by default, signals the end of the row, not the end of a line in a cell.
2. However, there is an exception to this rule: If a cell starts with a TEX-group (this is, with {), then inside this first group the \\ command retains the meaning of "end of line" character. Note that this special rule works only for the first group in a cell and this group must be at the beginning.

The net effect of these rules is the following: Normally, \\ is an end-of-row indicator; if you want to use it as an end-of-line indicator in a cell, just put the whole cell in curly braces. The following example illustrates the difference:

row 1	upper line
lower line	
row 2	hmm

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & upper line \\ lower line \\
    row 2 & hmm \\
  };
\end{tikzpicture}
```

row 1	upper line lower line
row 2	hmm

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of nodes,nodes={text width=16mm,draw}]
  {
    row 1 & {upper line \\ lower line} \\
    row 2 & hmm \\
  };
\end{tikzpicture}
```

Note that this system is not fool-proof. If you write things like `a&b{c\\d}\\` in a matrix of nodes, an error will result (because the second cell did not start with a brace, so \\ retained its normal meaning and, thus, the second cell contained the text `b{c`, which is not balanced with respect to the number of braces).

59.3 Delimiters

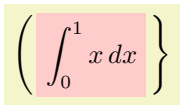
Delimiters are parentheses or braces to the left and right of a formula or a matrix. The `matrix` library offers options for adding such delimiters to a matrix. However, delimiters can actually be added to any node that has the standard anchors `north`, `south`, `north west` and so on. In particular, you can add delimiters to any `rectangle` box. They are implemented by "measuring the height" of the node and then adding a delimiter of the correct size to the left or right using some after node magic.

`/tikz/left delimiter=`⟨*delimiter*⟩ (no default)

This option can be given to a any node that has the standard anchors `north`, `south` and so on. The ⟨*delimiter*⟩ can be any delimiter that is acceptable to TEX's `\left` command.

$$\left(\begin{array}{ccc} a_8 & a_1 & a_6 \\ a_3 & a_5 & a_7 \\ a_4 & a_9 & a_2 \end{array}\right\}$$

```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```



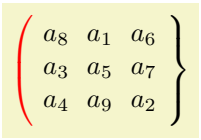
```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \node [fill=red!20,left delimiter=(,right delimiter=\}]
    {$\displaystyle\int_0^1 x\,dx$};
\end{tikzpicture}
```

`/tikz/every delimiter` (style, initially empty)

This style is executed for every delimiter. You can use it to shift or color delimiters or do whatever.

`/tikz/every left delimiter` (style, initially empty)

This style is additionally executed for every left delimiter.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  [every left delimiter/.style={red,xshift=1ex},
   every right delimiter/.style={xshift=-1ex}]
  \matrix [matrix of math nodes,left delimiter=(,right delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

`/tikz/right delimiter=`⟨*delimiter*⟩ (no default)

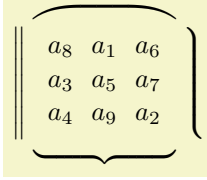
Works as above.

`/tikz/every right delimiter` (style, initially empty)

Works as above.

`/tikz/above delimiter=`⟨*delimiter*⟩ (no default)

This option allows you to add a delimiter above the node. It is implemented by rotating a left delimiter.



```
\usetikzlibrary {matrix}
\begin{tikzpicture}
  \matrix [matrix of math nodes,%
           left delimiter=\|,right delimiter=\rmoustache,%
           above delimiter=(,below delimiter=\}]
  {
    a_8 & a_1 & a_6 \\
    a_3 & a_5 & a_7 \\
    a_4 & a_9 & a_2 \\
  };
\end{tikzpicture}
```

`/tikz/every above delimiter` (style, initially empty)

Works as above.

`/tikz/below delimiter=`⟨*delimiter*⟩ (no default)

Works as above.

`/tikz/every below delimiter` (style, initially empty)

Works as above.

60 Mindmap Drawing Library

Ti*k*Z Library mindmap

```
\usetikzlibrary{mindmap} % LaTeX and plain TeX
\usetikzlibrary[mindmap] % ConTeXt
```

This packages provides styles for drawing mindmap diagrams.

60.1 Overview

This library is intended to make the creation of mindmaps or concept maps easier. A *mindmap* is a graphical representation of a concept together with related concepts and annotations. Mindmaps are, essentially, trees, possibly with a few extra edges added, but they are usually drawn in a special way: The root concept is placed in the middle of the page and is drawn as a huge circle, ellipse, or cloud. The related concepts then "leave" this root concept via branch-like tendrils.

The `mindmap` library of Ti*k*Z produces mindmaps that look a bit different from the standard mindmaps: While the big root concept is still a circle, related concepts are also depicted as (smaller) circles. The related concepts are linked to the root concept via organic-looking connections. The overall effect is visually rather pleasing, but readers may not immediately think of a mindmap when they see a picture created with this library.

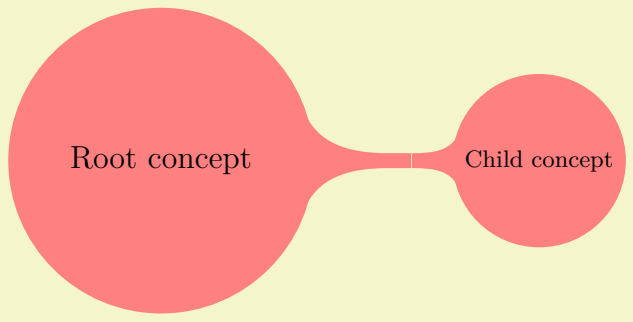
Although it is not strictly necessary, you will usually create mindmaps using Ti*k*Z's tree mechanism and some of the styles and macros of the package work best when used inside trees. However, it is still possible and sometimes necessary to treat parts of a mindmap as a graph with arbitrary edges and this is also possible.

60.2 The Mindmap Style

Every mindmap should be put in a scope or a picture where the `mindmap` style is used. This style installs some internal settings.

/tikz/mindmap (style, no value)

Use this style with all pictures or at least scopes that contain a mindmap. It installs a whole bunch of settings that are useful for drawing mindmaps.

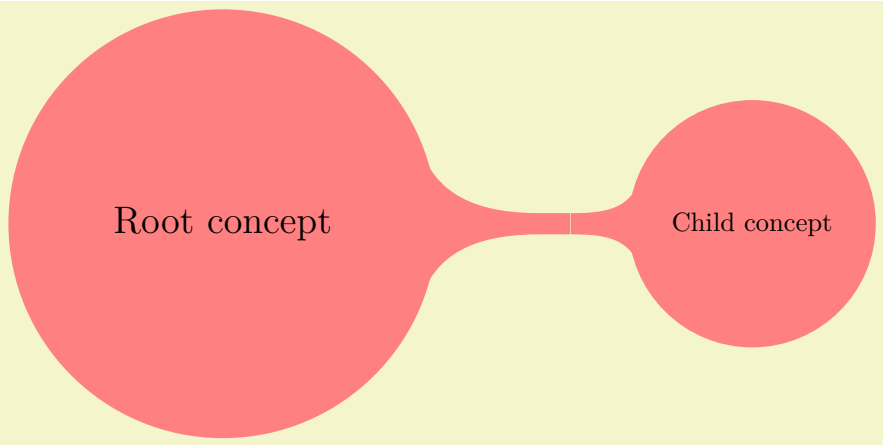



```
\usetikzlibrary {mindmap}
\tikz[mindmap,concept color=red!50]
  \node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

The sizes of concepts are predefined in such a way that a medium-size mindmap will fit on an A4 page (more or less).

/tikz/every mindmap (style, no value)

This style is included by the `mindmap` style. Change this style to add special settings to your mindmaps.




```
\usetikzlibrary {mindmap}
\tikz[large mindmap,concept color=red!50]
  \node [concept] {Root concept}
    child[grow=right] {node[concept] {Child concept}};
```

Remark: Note that `mindmap` redefines `font` sizes and `sibling angle` depending on the current concept level (i.e. inside of `level 1 concept`, `level 2 concept` etc.). Thus, if you need to redefine these variables, use

`level 1 concept/.append style={font=\small}`

or

`level 2 concept/.append style={sibling angle=90}`

after the `mindmap` style.

`/tikz/small mindmap` (style, no value)

This style includes the `mindmap` style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A5 page (A5 pages are half as large as A4 pages). Mindmaps with `small mindmap` will also fit onto a standard frame of the `beamer` package.

`/tikz/large mindmap` (style, no value)

This style includes the `mindmap` style, but additionally changes the default size of concepts, fonts and distances so that a medium-sized mindmap will fit on an A3 page (A3 pages are twice as large as A4 pages).

`/tikz/huge mindmap` (style, no value)

This style causes concepts to be even bigger and it is best used with A2 paper and above.

60.3 Concepts Nodes

The basic entities of mindmaps are called *concepts* in Ti*k*Z. A concept is a node of style `concept` and it must be circular for some of the connection macros to work.

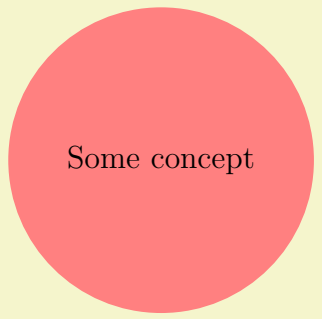
60.3.1 Isolated Concepts

The following styles influence how isolated concepts are rendered:

`/tikz/concept` (style, no value)

This style should be used with all nodes that are concepts, although some styles like `extra concept` install this style automatically.

Basically, this style makes the concept node circular and installs a uniform color called `concept color`, see below. Additionally, the style `every concept` is called.




```
\usetikzlibrary {mindmap}
\tikz[mindmap,concept color=red!50] \node [concept] {Some concept};
```

/tikz/every concept (style, no value)

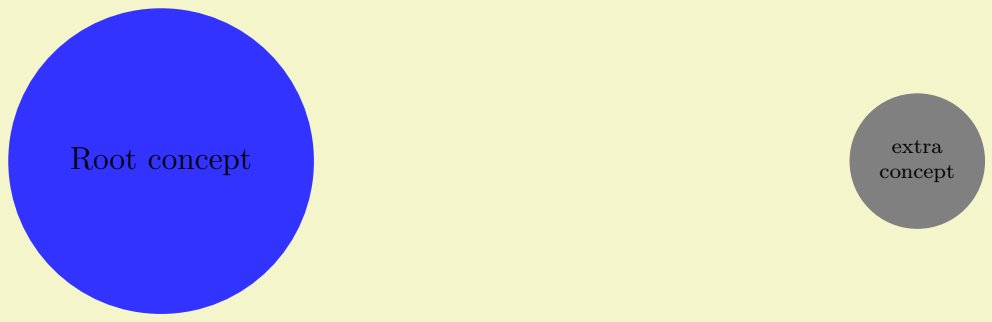
In order to change the appearance of concept nodes, you should change this style. Note, however, that the color of a concept should be uniform for some of the connection bar stuff to work, so you should not change the color or the draw/fill state of concepts using this option. It is mostly useful for changing the text color and font.

/tikz/concept color=⟨*color*⟩ (no default)

This option tells Ti*k*Z which color should be used for filling and stroking concepts. The difference between this option and just setting `every concept` to the desired color is that this option allows Ti*k*Z to keep track of the colors used for concepts. This is important when you *change* the color between two connected concepts. In this case, Ti*k*Z can automatically create a shading that provides a smooth transition between the old and the new concept color; we will come back to this in the next section.

/tikz/extra concept (style, no value)

This style is intended for concepts that are not part of the "mindmap tree", but stand beside it. Typically, they will have a subdued color or be smaller. In order to have these concepts appear in a uniform way and in order to indicate in the code that these concepts are additional, you can use this style.




```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[mindmap,concept color=blue!80]
  \node [concept]                {Root concept};
  \node [extra concept] at (10,0) {extra concept};
\end{tikzpicture}
```

/tikz/every extra concept (style, no value)

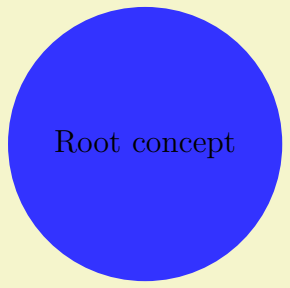
Change this style to change the appearance of extra concepts.

60.3.2 Concepts in Trees

As pointed out earlier, Ti*k*Z assumes that your mindmap is built using the `child` facilities of Ti*k*Z. There are numerous options that influence how concepts are rendered at the different levels of a tree.

/tikz/root concept (style, no value)

This style is used for the roots of mindmap trees. By adding something to this, you can change how the root of a mindmap will be rendered.

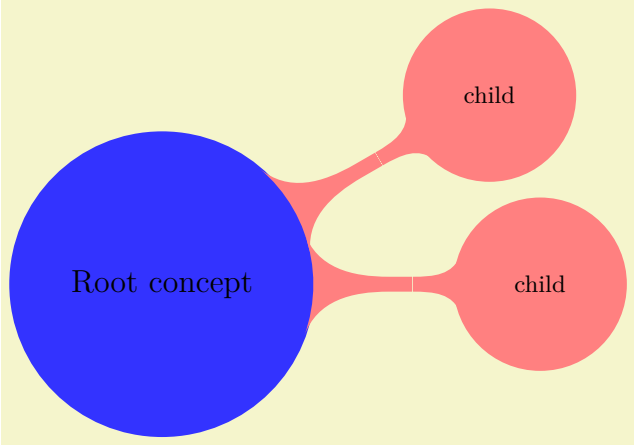



```
\usetikzlibrary {mindmap}
\tikz
  [root concept/.append style={concept color=blue!80,minimum size=3.5cm},
   mindmap]
  \node [concept] {Root concept};
```

Note that styles like `large mindmap` redefine these styles, so you should add something to this style only inside the picture.

/tikz/level 1 concept (style, no value)

The `mindmap` style adds this style to the `level 1` style. This means that the first level children of a mindmap tree will use this style.




```
\usetikzlibrary {mindmap}
\tikz
  [root concept/.append style={concept color=blue!80},
   level 1 concept/.append style={concept color=red!50},
   mindmap]
  \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0 ] {node[concept] {child}};
```

/tikz/level 2 concept (style, no value)

Works like `level 1 concept`, only for second level children.

/tikz/level 3 concept (style, no value)

Works like `level 1 concept`.

/tikz/level 4 concept (style, no value)

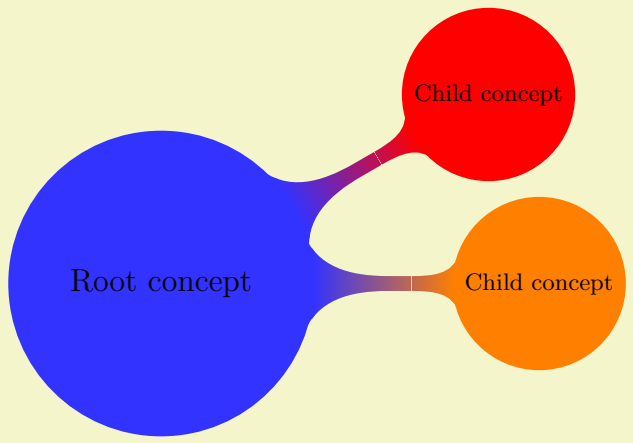
Works like `level 1 concept`. Note that there are no fifth and higher level styles, you need to modify `level 5` directly in such cases.

/tikz/concept color=⟨*color*⟩ (no default)

We saw already that this option is used to change the color of concepts. We now have a look at its effect when used on child nodes of a concept. Normally, this option simply changes the color of the children.

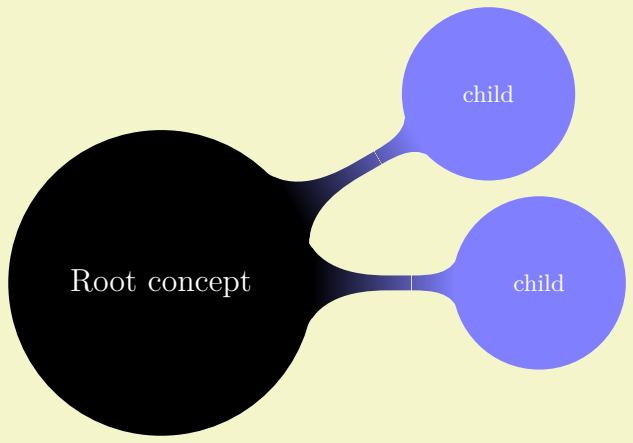
However, when the option is given as an option to the `child` operation (and not to the `node` operation and also not as an option to all children via the `level 1` style), Ti*k*Z will smoothly change the concept color from the parent's color to the color of the child concept.

Here is an example:




```
\usetikzlibrary {mindmap}
\tikz[mindmap,concept color=blue!80]
  \node [concept] {Root concept}
    child[concept color=red,grow=30] {node[concept] {Child concept}}
    child[concept color=orange,grow=0]  {node[concept] {Child concept}};
```

In order to have a concept color which changes with the hierarchy level, a tiny bit of magic is needed:




```
\usetikzlibrary {mindmap}
\tikz[mindmap,text=white,
      root concept/.style={concept color=blue},
      level 1 concept/.append style=
        {every child/.style={concept color=blue!50}}]
  \node [concept] {Root concept}
    child[grow=30] {node[concept] {child}}
    child[grow=0 ] {node[concept] {child}};
```

60.4 Connecting Concepts

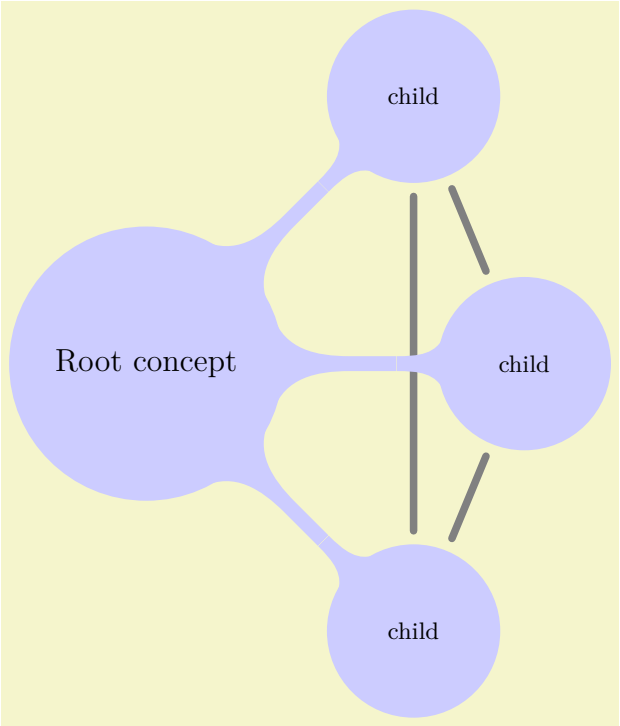
60.4.1 Simple Connections

The easiest way to connect two concepts is to draw a line between them. In order to give such lines a consistent appearance, it is recommendable to use the following style when drawing such lines:

`/tikz/concept connection` (style, no value)

This style can be used for lines between two concepts. Feel free to redefine this style.

A problem arises when you need to connect concepts after the main mindmap has been drawn. In this case you will want the connection lines to lie *behind* the main mindmap. However, you can draw the lines only after the coordinates of the concepts have been determined. In this case you should place the connecting lines on a background layer as in the following example:




```
\usetikzlibrary {backgrounds,mindmap}
\begin{tikzpicture}
  [root concept/.append style={concept color=blue!20,minimum size=2cm},
   level 1 concept/.append style={sibling angle=45},
   mindmap]
  \node [concept] {Root concept}
    [clockwise from=45]
    child { node[concept] (c1) {child}}
    child { node[concept] (c2) {child}}
    child { node[concept] (c3) {child}};
  \begin{pgfonlayer}{background}
    \draw [concept connection]  (c1) edge (c2)
                                     edge (c3)
                                (c2) edge (c3);
  \end{pgfonlayer}
\end{tikzpicture}
```

60.4.2 The Circle Connection Bar Decoration

Instead of a simple line between two concepts, you can also add a bar between the two nodes that has slightly organic ends. These bars are also used by default as the edges from parents in the mindmap tree.

For the drawing of the bars a special decoration is used, which is defined in the `mindmap` library:

Decoration `circle connection bar`

This decoration can be used to connect two circles. The start of the to-be-decorated path should lie on the border of the first circle, the end should lie on the border of the second circle. The following two decoration keys should be initialized with the sizes of the circles:

- `start radius`
- `end radius`

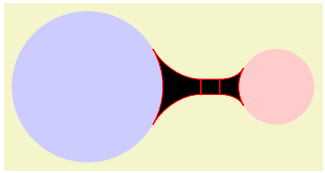
Furthermore, the following two decoration keys influence the decoration:

- `amplitude`

- `angle`

The decoration turns a straight line into a path that starts on the border of the first circle at the specified angle relative to the line connecting the centers of the circles. The path then changes into a rectangle whose thickness is given by the amplitude. Finally, the path ends with the same angles on the second circle.

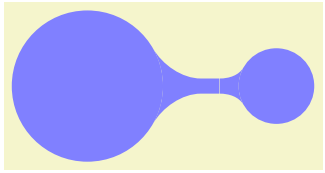
Here is an example that should make this clearer:



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
    [decoration={start radius=1cm,end radius=.5cm,amplitude=2mm,angle=30}]
  \fill[blue!20] (0,0)   circle [radius=1cm];
  \fill[red!20]  (2.5,0) circle [radius=.5cm];

  \filldraw [draw=red,fill=black,
             decorate,decoration=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

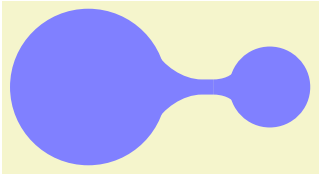
As can be seen, the decorated path consists of three parts and is not really useful for drawing. However, if you fill the decorated path only, and if you use the same color as for the circles, the result is better.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [blue!50,decoration={start radius=1cm,
                       end radius=.5cm,amplitude=2mm,angle=30}]
  \fill (0,0)   circle [radius=1cm];
  \fill (2.5,0) circle [radius=.5cm];

  \fill [decorate,decoration=circle connection bar] (1,0) -- (2,0);
\end{tikzpicture}
```

In the above example you may notice the small white line between the circles and the decorated path. This is due to rounding errors. Unfortunately, for larger distances, the errors can accumulate quite strongly, especially since Ti*k*Z and TEX are not very good at computing square roots. For this reason, it is a good idea to make the circles slightly larger to cover up such problems. When using nodes of shape `circle`, you can just add the `draw` option with a `line width` of one or two points (for very large distances you may need line width up to 4pt).



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [blue!50,decoration={start radius=1cm,
                       end radius=.5cm,amplitude=2mm,angle=30}]
  \fill (0,0)   circle [radius=1cm+1pt];
  \fill (2.4,0) circle [radius=.5cm+1pt];

  \fill [decorate,decoration=circle connection bar] (1,0) -- (1.9,0);
\end{tikzpicture}
```

60.4.3 The Circle Connection Bar To-Path

The `circle connection bar` decoration is a bit complicated to use. Especially specifying the radii is quite bothersome (the amplitude and the angle can be set once and for all). For this reason, the `mindmap` library defines a special to-path that performs the necessary computations for you.

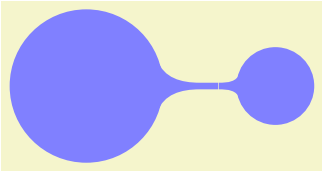
`/tikz/circle connection bar` (style, no value)

This style installs a rather involved to-path. Unlike normal to-paths, this path requires that the start and the target of the to-path are named nodes of shape `circle` – if this is not the case, this path will produce errors.

Assuming that the start and the target are circles, the to-path will first compute the radii of these circles (by measuring the distance from the `center` anchor to some anchor on the border) and will set the `start circle` keys accordingly. Next, the `fill` option is set to the `concept color` while `draw=none` is set. The decoration is set to `circle connection bar`. Finally, the following style is included:

`/tikz/every circle connection bar` (style, no value)

Redefine this style to change the appearance of circle connection bar to-paths.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[concept color=blue!50,blue!50,outer sep=0pt]
  \node (n1) at (0,0)   [circle,minimum size=2cm,fill,draw,thick] {};
  \node (n2) at (2.5,0) [circle,minimum size=1cm,fill,draw,thick] {};

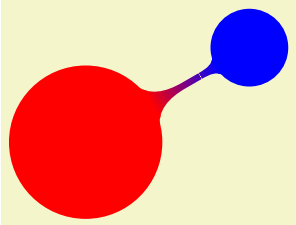
  \path (n1) to[circle connection bar] (n2);
\end{tikzpicture}
```

Note that it is not a good idea to have more than one `to` operation together with the option `circle connection bar` in a single `\path`. Use the `edge` operation, instead, for creating multiple connections and this operation creates a new scope for each edge.

In a mindmap we sometimes want colors to change from one concept color to another. Then, the connection bar should, ideally, consist of a smooth transition between these two colors. Getting this right using shadings is a bit tricky if you try this "by hand", so the `mindmap` library provides a special option for facilitating this procedure.

`/tikz/circle connection bar switch color=from (`⟨*first color*⟩`) to (`⟨*second color*⟩`)` (no default)

This style works similarly to the `circle connection bar`. The only difference is that instead of filling the path with a single color a shading is used.



```
\usetikzlibrary {mindmap}
\begin{tikzpicture}[outer sep=0pt]
  \node (n1) at (0,0)    [circle,minimum size=2cm,fill,draw,thick,red] {};
  \node (n2) at (30:2.5) [circle,minimum size=1cm,fill,draw,thick,blue] {};

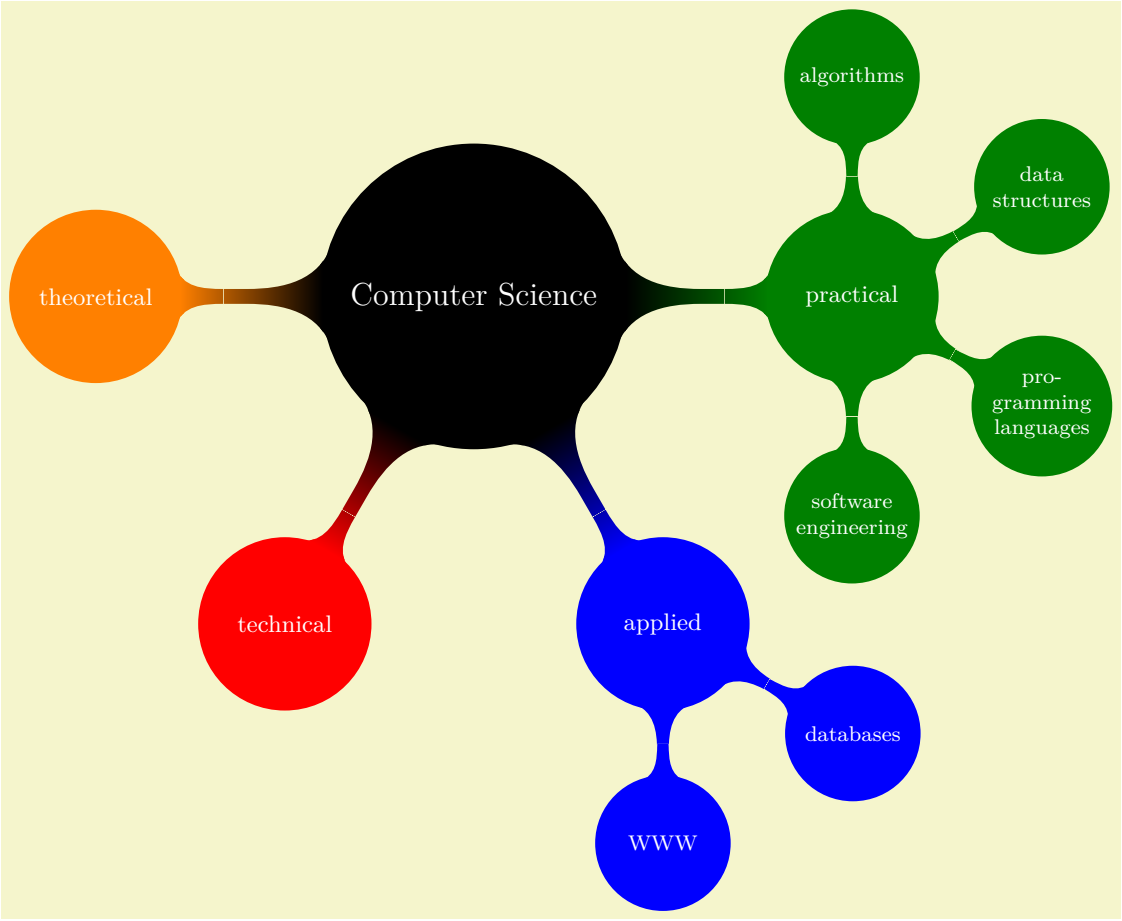
  \path (n1) to[circle connection bar switch color=from (red) to (blue)] (n2);
\end{tikzpicture}
```

60.4.4 Tree Edges

Most of the time, concepts in a mindmap are connected automatically when the mindmap is built as a tree. The reason is that the `mindmap` installs a `circle connection bar` path as the `edge from parent path`. Also, the `mindmap` option takes care of things like setting the correct `draw` and `outer sep` settings and some other stuff.

In detail, the `mindmap` option sets the `edge from parent path` to a path that uses the to-path `circle connection bar` to connect the parent node and the child node. The `concept color` option (locally) changes this by using `circle connection bar switch color` instead with the from-color set to the old (parent's) concept color and the to-color set to the new (child's) concept color. This means that when you provide the `concept color` option to a `child` command, the color will change from the parent's concept color to the specified color.

Let us now build a tree that way. Please note that we pass the `concept color` to the respective `child` and not to a `node` under it.



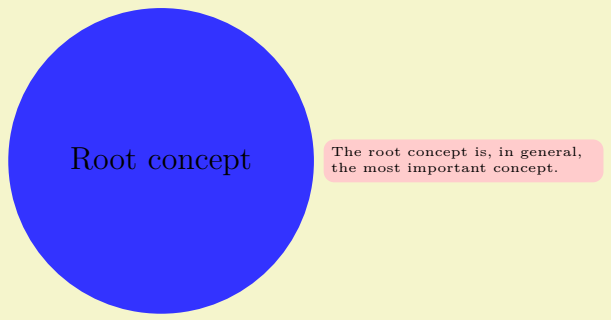

```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  \path[mindmap,concept color=black,text=white]
    node[concept] {Computer Science}
    [clockwise from=0]
    % note that `sibling angle' can only be defined in
    % `level 1 concept/.append style={}'
    child[concept color=green!50!black] {
      node[concept] {practical}
      [clockwise from=90]
      child { node[concept] {algorithms} }
      child { node[concept] {data structures} }
      child { node[concept] {pro\-gramming languages} }
      child { node[concept] {software engineer\-ing} }
    }
    % note that the `concept color' is passed to the `child'(!)
    child[concept color=blue] {
      node[concept] {applied}
      [clockwise from=-30]
      child { node[concept] {databases} }
      child { node[concept] {WWW} }
    }
    child[concept color=red] { node[concept] {technical} }
    child[concept color=orange] { node[concept] {theoretical} };
\end{tikzpicture}
```

60.5 Adding Annotations

An *annotation* is some text outside a mindmap that, unlike an extra concept, simply explains something in the mindmap. The following style is mainly intended to help readers of the code see that a node in an annotation node.

/tikz/annotation (style, no value)

This style indicates that a node is an annotation node. It includes the style `every annotation`, which allows you to change this style in a convenient fashion.




```
\usetikzlibrary {mindmap}
\begin{tikzpicture}
  [mindmap,concept color=blue!80,
  every annotation/.style={fill=red!20}]
  \node [concept] (root)  {Root concept};

  \node [annotation,right] at (root.east)
  {The root concept is, in general, the most important concept.};
\end{tikzpicture}
```

`/tikz/every annotation` (style, no value)

This style is included by `annotation`.

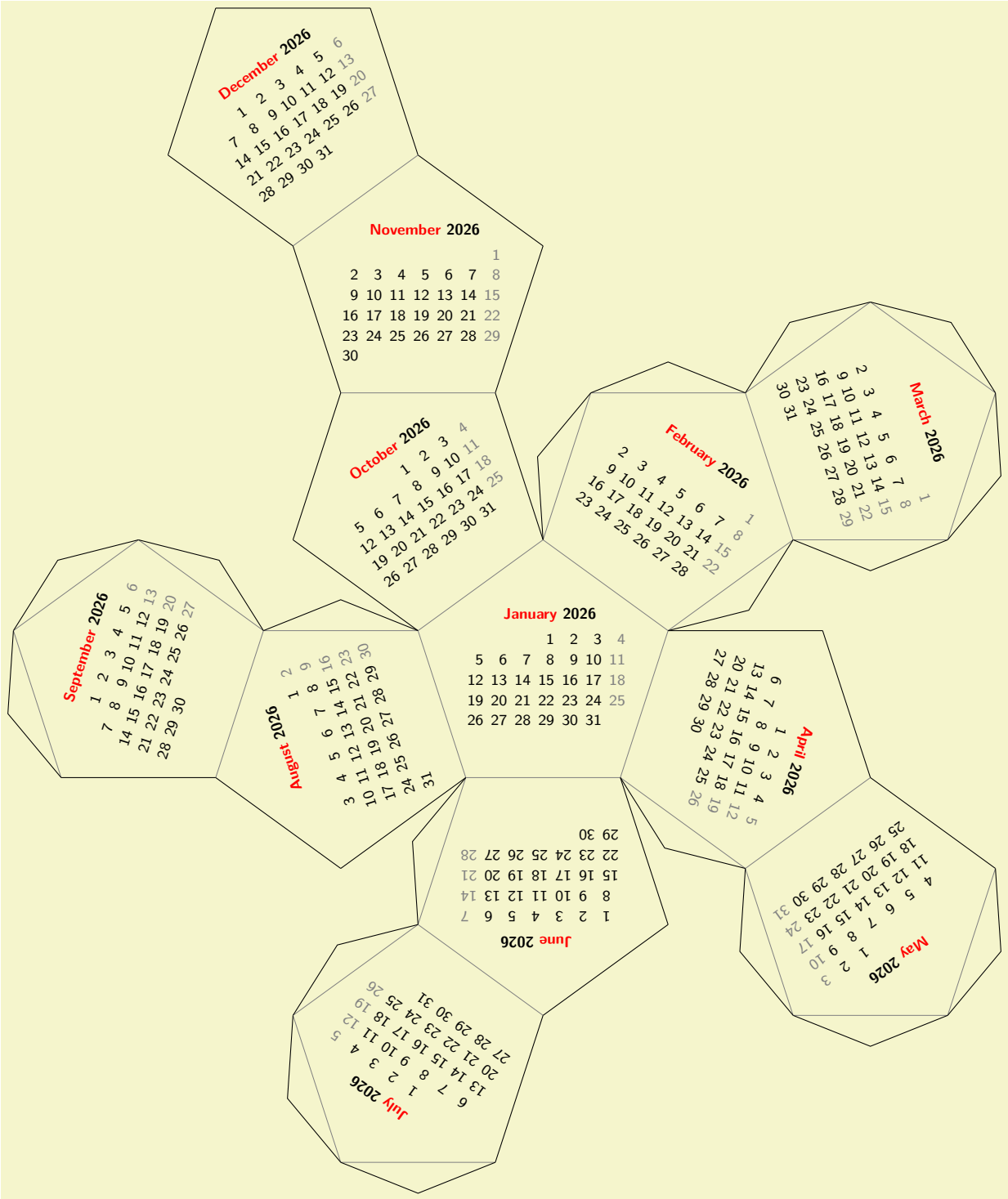
61 Paper-Folding Diagrams Library

TikZ Library folding

```
\usetikzlibrary{folding} % LaTeX and plain TeX
\usetikzlibrary[folding] % ConTeXt
```

This library defines pic types for creating paper-folding diagrams. Many thanks to Nico van Cleemput for providing most of the code.

Here is a big example that produces a diagram for a calendar:



```
\usetikzlibrary {calendar,folding}
\sffamily\scriptsize
\tikz \pic [
  transform shape,
  every calendar/.style={
    at={(-8ex,4ex)},
    week list,
    month label above centered,
    month text=\bfseries\textcolor{red}{\%mt} \%y0,
    if={(Sunday) [black!50]}
  },
  folding line length=2.5cm,
  face 1={ \calendar [dates=\the\year-01-01 to \the\year-01-last];},
  face 2={ \calendar [dates=\the\year-02-01 to \the\year-02-last];},
  face 3={ \calendar [dates=\the\year-03-01 to \the\year-03-last];},
  face 4={ \calendar [dates=\the\year-04-01 to \the\year-04-last];},
  face 5={ \calendar [dates=\the\year-05-01 to \the\year-05-last];},
  face 6={ \calendar [dates=\the\year-06-01 to \the\year-06-last];},
  face 7={ \calendar [dates=\the\year-07-01 to \the\year-07-last];},
  face 8={ \calendar [dates=\the\year-08-01 to \the\year-08-last];},
  face 9={ \calendar [dates=\the\year-09-01 to \the\year-09-last];},
  face 10={\calendar [dates=\the\year-10-01 to \the\year-10-last];},
  face 11={\calendar [dates=\the\year-11-01 to \the\year-11-last];},
  face 12={\calendar [dates=\the\year-12-01 to \the\year-12-last];}
] {dodecahedron folding};
```

The foldings are sorted by number of faces.

Pic type `tetrahedron folding`

This pic type draws a folding diagram for a tetrahedron. The following keys influence the pic:

`/tikz/folding line length=`⟨*dimension*⟩ (no default)

Sets the length of the base line for folding. For the dodecahedron this is the length of all the sides of the pentagons.

`/tikz/face 1=`⟨*code*⟩ (no default)

The ⟨*code*⟩ is executed for the first face of the dodecahedron. When it is executed, the coordinate system will have been shifted and rotated such that it lies at the middle of the first face of the dodecahedron.

`/tikz/face 2=`⟨*code*⟩ (no default)

Same as `face 1`, but for the second face.

`/tikz/face 3=`⟨*code*⟩ (no default)

`/tikz/face 4=`⟨*code*⟩ (no default)

There are further similar options for more faces (for commands shown later).

Here is a simple example:




```
\usetikzlibrary {folding}
\tikz \pic [
  transform shape,
  folding line length=6mm,
  face 1={ \node[red] {1};},
  face 2={ \node      {2};},
  face 3={ \node      {3};},
  face 4={ \node      {4};}
] {tetrahedron folding};
```

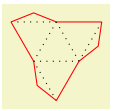
The appearance of the cut and folding lines can be influenced using the following styles:

`/tikz/every cut` (style, initially empty)

Executed for every line that should be cut using scissors.

`/tikz/every fold` (style, initially `help lines`)

Executed for every line that should be folded.



```
\usetikzlibrary {folding}
\tikz \pic[
  every cut/.style=red,
  every fold/.style=dotted,
  folding line length=6mm
] { tetrahedron folding };
```

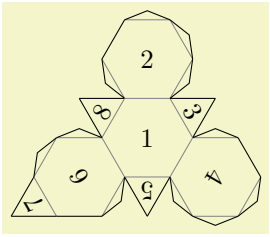
There is one style that is mainly useful for the present documentation:

`/tikz/numbered faces` (style, no value)

Sets `face` ⟨*i*⟩ to `\node {`⟨*i*⟩`};` for all *i*.

Pic type `tetrahedron truncated folding`

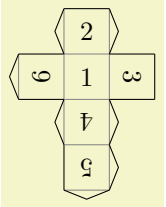
A folding of a truncated tetrahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { tetrahedron truncated folding };
```

Pic type `cube folding`

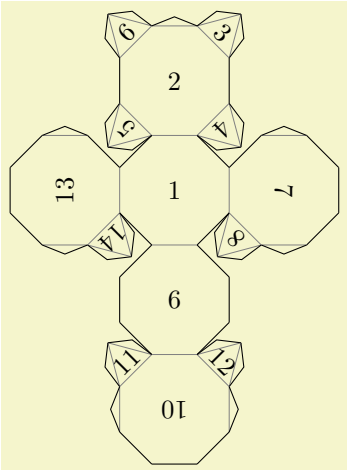
A folding of a cube.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { cube folding };
```

Pic type `cube truncated folding`

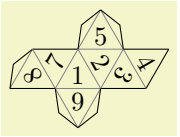
A folding of a truncated cube.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { cube truncated folding };
```

Pic type `octahedron folding`

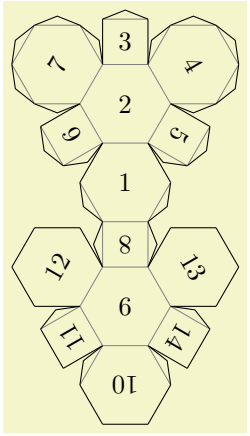
A folding of an octahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { octahedron folding };
```

Pic type `octahedron folding`

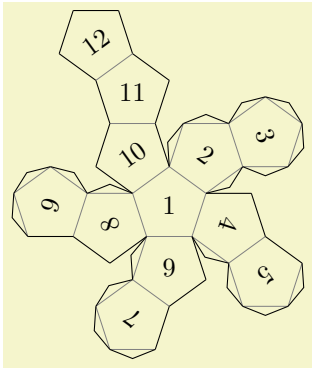
A folding of a truncated octahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { octahedron truncated folding };
```

Pic type `dodecahedron folding`

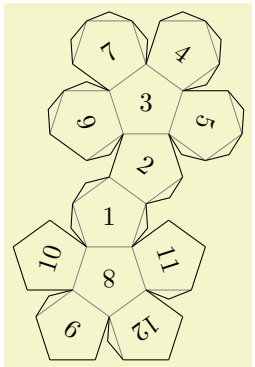
A folding of a dodecahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { dodecahedron folding };
```

Pic type `dodecahedron' folding`

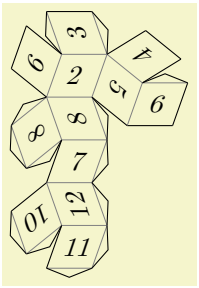
This is an alternative folding of a dodecahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { dodecahedron' folding };
```

Pic type dodecahedron rhombic folding

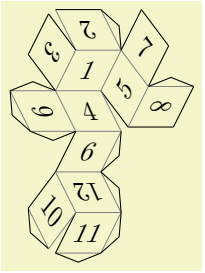
A rhombic dodecahedron, based on square-of-two rhombi.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { dodecahedron rhombic folding };
```

Pic type dodecahedron golden folding

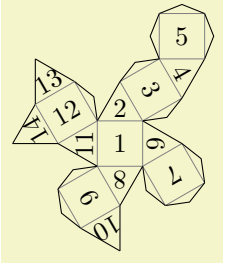
An alternative rhombic dodecahedron, flat laying, based on golden rhombi.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { dodecahedron golden folding };
```

Pic type cuboctahedron folding

A folding of a cuboctahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { cuboctahedron folding };
```

Pic type cuboctahedron truncated folding

A folding of a truncated cuboctahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { cuboctahedron truncated folding };
```

Pic type `icosahedron folding`

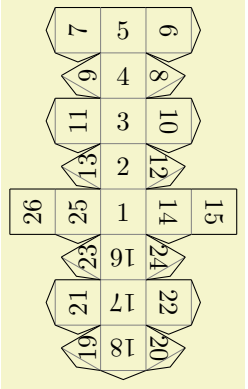
A folding of an icosahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { icosahedron folding };
```

Pic type `rhombicuboctahedron folding`

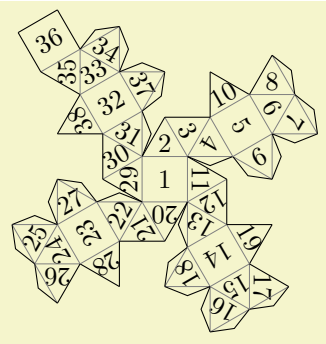
A folding of an rhombicuboctahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { rhombicuboctahedron folding };
```

Pic type `snub cube folding`

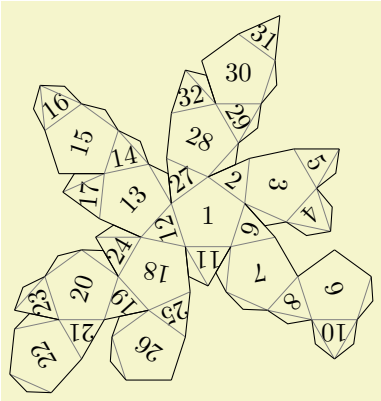
A folding of a snub cube.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { snub cube folding };
```

Pic type `icosidodecahedron folding`

A folding of an icosidodecahedron.




```
\usetikzlibrary {folding}
\tikz \pic [folding line length=6mm, numbered faces, transform shape]
  { icosidodecahedron folding };
```

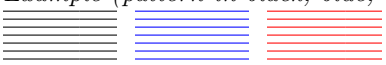
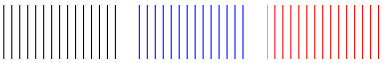
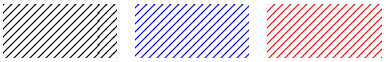
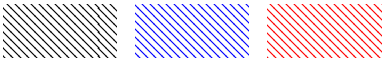
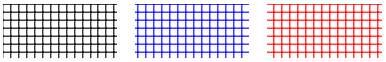
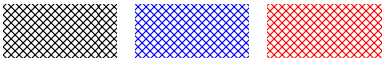
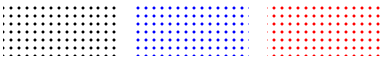
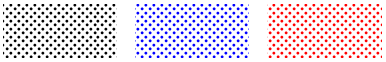
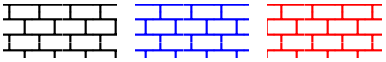
62 Pattern Library

Ti*k*Z Library patterns

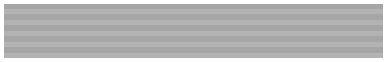
```
\usepgflibrary{patterns}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[patterns]  % ConTeXt when using pure PGF
\usetikzlibrary{patterns} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[patterns] % ConTeXt when using TikZ
```

The package defines patterns for filling areas.

62.1 Form-Only Patterns

Pattern name	*Example (pattern in black, blue, and red on faded checkerboard)*
horizontal lines	
vertical lines	
north east lines	
north west lines	
grid	
crosshatch	
dots	
crosshatch dots	
fivepointed stars	
sixpointed stars	
bricks	
checkerboard	

62.2 Inherently Colored Patterns

Pattern name	*Example*
`checkerboard light gray`	
`horizontal lines light gray`	
`horizontal lines gray`	
`horizontal lines dark gray`	
`horizontal lines light blue`	
`horizontal lines dark blue`	
`crosshatch dots gray`	
`crosshatch dots light steel blue`	

62.3 User-Defined Patterns

by Mark Wibrow

TikZ Library `patterns.meta`

```
\usepgflibrary{patterns.meta}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[patterns.meta]  % ConTeXt when using pure PGF
\usetikzlibrary{patterns.meta} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[patterns.meta] % ConTeXt when using TikZ
```

Define your own patterns with a syntax similar to `arrows.meta`.

Caveat: This library is currently experimental and might change without notice. There are some known shortcomings that will hopefully be fixed in the future.

`\pgfdeclarepattern{`⟨*config*⟩`}`

This command is used to declare a new pattern. In contrast to the normal patterns and in the spirit of `arrows.meta` this command takes a list of keys and values to define the pattern. The following keys are available:

`/pgf/patterns/name=`⟨*name*⟩ (no default)

The name of the pattern by which it can be used later on.

`/pgf/patterns/type=`⟨*type*⟩ (default `uncolored`)

The type of the pattern maps to what was called "form only" and "inherently colored" in the language of the normal patterns. The available choices are:

- `uncolored` the pattern will obey the surrounding color.
- `colored` the pattern will have an intrinsic color.
- `form only` synonym for `uncolored`
- `inherently colored` synonym for `colored`

`/pgf/patterns/x=`⟨*dimension*⟩ (default `1cm`)

Unit vector of the coordinate system in the x-direction.

`/pgf/patterns/y=`⟨*dimension*⟩ (default `1cm`)

Unit vector of the coordinate system in the y-direction.

/pgf/patterns/parameters=⟨*comma separated list*⟩ (default empty)

A list of parameters that are passed to the pattern. This is usually a list of TEX macros. It is very important that these macros are fully expandable because the values they hold are being used for deduplication in the PDF file.

/pgf/patterns/defaults=⟨*comma separated list*⟩ (default empty)

This list holds default assignments to the parameters passed to the pattern. The default keys can then be found under the `/pgf/pattern keys/` prefix.

/pgf/patterns/bottom left=⟨*pgfpoint*⟩ (no default)

Bottom left corner of the pattern's bounding box, e.g. `\pgfqpoint{-.1pt}{-.1pt}`.

/pgf/patterns/top right=⟨*pgfpoint*⟩ (no default)

Top right corner of the pattern's bounding box, e.g. `\pgfqpoint{3.1pt}{3.1pt}`.

/pgf/patterns/tile size=⟨*pgfpoint*⟩ (no default)

Width and height of a single of the pattern as a PGF point specification, i.e. the x coordinate is the width and the y coordinate is the height, e.g. `\pgfqpoint{3pt}{3pt}`.

/pgf/patterns/tile transformation=⟨*pgftransformation*⟩ (default empty)

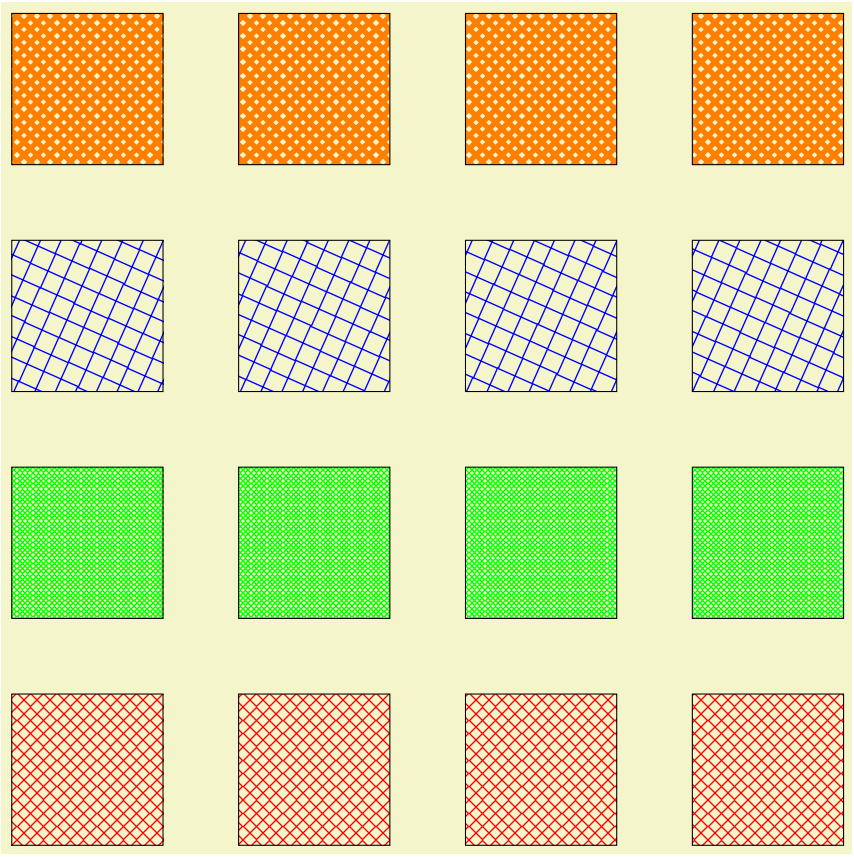
A PGF transformation, e.g. `\pgftransformrotate{30}`.

/pgf/patterns/code=⟨*code*⟩ (no default)

The code should be PGF code that can be protocolled. It should not contain any color code or nodes.

/pgf/patterns/set up code=⟨*code*⟩ (default empty)

This code can be set if parameters have to be preprocessed before the actual pattern code can be run.



```
\usetikzlibrary {patterns.meta}
\pgfdeclarepattern{
  name=hatch,
  parameters={\hatchsize,\hatchangle,\hatchlinewidth},
  bottom left={\pgfpoint{-.1pt}{-.1pt}},
  top right={\pgfpoint{\hatchsize+.1pt}{\hatchsize+.1pt}},
  tile size={\pgfpoint{\hatchsize}{\hatchsize}},
  tile transformation={\pgftransformrotate{\hatchangle}},
  code={
    \pgfsetlinewidth{\hatchlinewidth}
    \pgfpathmoveto{\pgfpoint{-.1pt}{-.1pt}}
    \pgfpathlineto{\pgfpoint{\hatchsize+.1pt}{\hatchsize+.1pt}}
    \pgfpathmoveto{\pgfpoint{-.1pt}{\hatchsize+.1pt}}
    \pgfpathlineto{\pgfpoint{\hatchsize+.1pt}{-.1pt}}
    \pgfusepath{stroke}
  }
}

\tikzset{
  hatch size/.store in=\hatchsize,
  hatch angle/.store in=\hatchangle,
  hatch line width/.store in=\hatchlinewidth,
  hatch size=5pt,
  hatch angle=0pt,
  hatch line width=.5pt,
}

\begin{tikzpicture}
\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=red]
    (\r*3,0) rectangle ++(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=green, hatch size=2pt]
    (\r*3,3) rectangle ++(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=blue, hatch size=10pt, hatch angle=21]
    (\r*3,6) rectangle ++(2,2);

\foreach \r in {1,...,4}
  \draw [pattern=hatch, pattern color=orange, hatch line width=2pt]
    (\r*3,9) rectangle ++(2,2);
\end{tikzpicture}
```

There are a couple of predefined PGF patterns which are similar to their normal counterparts. For all of these the `xshift` and `yshift` are applied *before* the rotation. If you want to rotate before shifting, just rotate in the drawing code.

Pattern `Lines`

The `Lines` pattern replaces four patterns: `horizontal lines`, `vertical lines`, `north east lines`, and `north west lines`. Unfortunately, due to the way the old patterns are constructed, namely that they are not simply related to each other by rotation, the `Lines` pattern cannot be used as a drop-in replacement.

However, the pattern options can be tuned to resemble the other versions closely. The available parameters are:

`/pgf/pattern keys/distance` (initially `3pt`)

Distance between lines.

`/pgf/pattern keys/angle` (initially `0`)

By default the lines are horizontal. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

`/pgf/pattern keys/xshift` (initially `0pt`)

Shifts the whole pattern in x-direction (before applying the rotation).

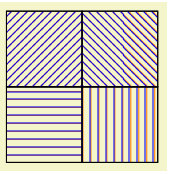
`/pgf/pattern keys/yshift` (initially `0pt`)

Shifts the whole pattern in y-direction (before applying the rotation).

`/pgf/pattern keys/line width` (initially `\the\pgflinewidth`)

Thickness of the lines.

The following settings can be used to reproduce the other `... lines` patterns.



```
\usetikzlibrary {patterns,patterns.meta}
\begin{tikzpicture}
  \draw[pattern={horizontal lines},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Lines[yshift=.5pt]},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={vertical lines},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Lines[angle=90,yshift=-.5pt]},pattern color=blue]
    (1,0) rectangle +(1,1);

  \draw[pattern={north east lines},pattern color=orange]
    (0,1) rectangle +(1,1);
  \draw[pattern={Lines[angle=45,distance={3pt/sqrt(2)}]},pattern color=blue]
    (0,1) rectangle +(1,1);

  \draw[pattern={north west lines},pattern color=orange]
    (1,1) rectangle +(1,1);
  \draw[pattern={Lines[angle=-45,distance={3pt/sqrt(2)}]},pattern color=blue]
    (1,1) rectangle +(1,1);
\end{tikzpicture}
```

Pattern `Hatch`

The `Hatch` pattern replaces the `grid` and `crosshatch` patterns. The `Hatch` pattern without options is a drop-in replacement for the `grid` pattern.

`/pgf/pattern keys/distance` (initially `3pt`)

Distance between crosses.

`/pgf/pattern keys/angle` (initially `0`)

By default the lines are horizontal and vertical. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

`/pgf/pattern keys/xshift` (initially `0pt`)

Shifts the whole pattern in x-direction (before applying the rotation).

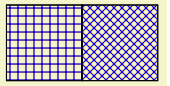
`/pgf/pattern keys/yshift` (initially `0pt`)

Shifts the whole pattern in y-direction (before applying the rotation).

`/pgf/pattern keys/line width` (initially `\the\pgflinewidth`)

Thickness of the lines.

The following settings can be used to reproduce the `grid` and `crosshatch` patterns.



```
\usetikzlibrary {patterns,patterns.meta}
\begin{tikzpicture}
  \draw[pattern={grid},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Hatch},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={crosshatch},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Hatch[angle=45,distance={3pt/sqrt(2)},xshift=.1pt]},
    pattern color=blue] (1,0) rectangle +(1,1);
\end{tikzpicture}
```

Pattern `Dots`

The `Dots` pattern replaces the `dots` and `crosshatch dots` patterns. The `Dots` pattern without options is a drop-in replacement for the `dots` pattern.

`/pgf/pattern keys/distance` (initially `3pt`)

Distance between dots.

`/pgf/pattern keys/angle` (initially `0`)

By default the lines are arranged on a regular grid. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

`/pgf/pattern keys/xshift` (initially `0pt`)

Shifts the whole pattern in x-direction (before applying the rotation).

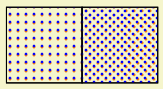
`/pgf/pattern keys/yshift` (initially `0pt`)

Shifts the whole pattern in y-direction (before applying the rotation).

`/pgf/pattern keys/radius` (initially `0.5pt`)

Radius of the dots.

The following settings can be used to reproduce the `dots` and `crosshatch dots` patterns.



```
\usetikzlibrary {patterns,patterns.meta}
\begin{tikzpicture}
  \draw[pattern={dots},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Dots},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={crosshatch dots},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Dots[angle=45,distance={3pt/sqrt(2)}]},
    pattern color=blue] (1,0) rectangle +(1,1);
\end{tikzpicture}
```

Pattern `Stars`

The `Stars` pattern replaces the `fivepointed stars` and `sixpointed stars` patterns. However, the stars of the `Stars` pattern are constructed in a fundamentally different fashion, so it can't be used as a drop-in replacement.

`/pgf/pattern keys/distance` (initially `3mm`)

Distance between stars.

`/pgf/pattern keys/angle` (initially `0`)

By default the stars are arranged on a regular grid. The whole pattern is rotated by this angle. The rotation angle is measured in the mathematically positive sense.

`/pgf/pattern keys/xshift` (initially `0pt`)

Shifts the whole pattern in x-direction (before applying the rotation).

`/pgf/pattern keys/yshift` (initially `0pt`)

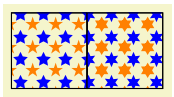
Shifts the whole pattern in y-direction (before applying the rotation).

`/pgf/pattern keys/radius` (initially `1mm`)

Outer radius of the enclosing circle of the stars.

`/pgf/pattern keys/points` (initially `5`)

Number of pointy ends of the stars.



```
\usetikzlibrary {patterns,patterns.meta}
\begin{tikzpicture}
  \draw[pattern={fivepointed stars},pattern color=orange]
    (0,0) rectangle +(1,1);
  \draw[pattern={Stars},pattern color=blue]
    (0,0) rectangle +(1,1);

  \draw[pattern={sixpointed stars},pattern color=orange]
    (1,0) rectangle +(1,1);
  \draw[pattern={Stars[points=6]},pattern color=blue]
    (1,0) rectangle +(1,1);
\end{tikzpicture}
```

\tikzdeclarepattern{⟨*config*⟩}

A pattern declared with `\pgfdeclarepattern` can only execute PGF code. This command extends the functionality to also allow Ti*k*Z code. All the same keys of `\pgfdeclarepattern` are valid, but some of them have been overloaded to give a more natural Ti*k*Z syntax.

/tikz/patterns/bottom left=⟨*point*⟩ (no default)

Instead of a PGF name point, this key takes a Ti*k*Z point, e.g. `(-.1,-.1)`.

/tikz/patterns/top right=⟨*point*⟩ (no default)

Instead of a PGF name point, this key takes a Ti*k*Z point, e.g. `(3.1,3.1)`.

/tikz/patterns/tile size=⟨*point*⟩ (no default)

Instead of a PGF name point, this key takes a Ti*k*Z point, e.g. `(3,3)`.

/tikz/patterns/tile transformation=⟨*transformation*⟩ (no default)

Instead of a PGF transformation, this key takes a list of keys and value and extracts the resulting transformation from them, e.g. `rotate=30`.

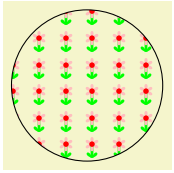
In addition to the overloaded keys, some new keys have been added.

/tikz/patterns/bounding box=⟨*point*⟩ and ⟨*point*⟩ (no default)

This is a shorthand to set the bounding box. It will assign the first point to `bottom left` and the second point to `top right`.

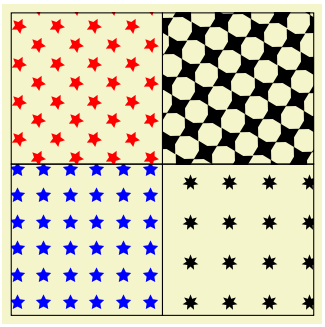
/tikz/patterns/infer tile bounding box=⟨*dimension*⟩ (default `0pt`)

Instead of specifying the bounding box by hand, you can ask Ti*k*Z to infer the size of the bounding box for you. The ⟨*dimension*⟩ parameter is padding that is added around the bounding box.



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=flower,
  type=colored,
  bottom left={(-.1pt,-.1pt)},
  top right={(10.1pt,10.1pt)},
  tile size={(10pt,10pt)},
  code={
    \tikzset{x=1pt,y=1pt}
    \path [draw=green] (5,2.5) -- (5, 7.5);
    \foreach \i in {0,60,...,300}
      \path [fill=pink, shift={(5,7.5)}, rotate=-\i]
        (0,0) .. controls ++(120:4) and ++(60:4) .. (0,0);
    \path [fill=red] (5,7.5) circle [radius=1];
    \foreach \i in {-45,45}
      \path [fill=green, shift={(5,2.5)}, rotate=-\i]
        (0,0) .. controls ++(120:4) and ++(60:4) .. (0,0);
  }
}

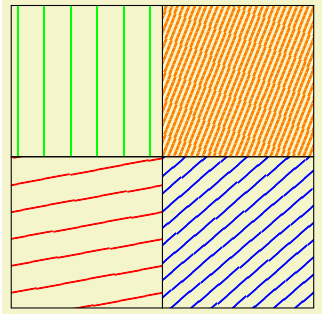
\tikz\draw [pattern=flower] circle [radius=1];
```



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=mystars,
  type=uncolored,
  bounding box={(-5pt,-5pt) and (5pt,5pt)},
  tile size={(\tikztilesize,\tikztilesize)},
  parameters={\tikzstarpoints,\tikzstarradius,\tikzstarrotate,\tikztilesize},
  tile transformation={rotate=\tikzstarrotate},
  defaults={
    points/.store in=\tikzstarpoints,points=5,
    radius/.store in=\tikzstarradius,radius=3pt,
    rotate/.store in=\tikzstarrotate,rotate=0,
    tile size/.store in=\tikztilesize,tile size=10pt,
  },
  code={
    \pgfmathparse{180/\tikzstarpoints}\let\a=\pgfmathresult
    \fill (90:\tikzstarradius) \foreach \i in {1,...,\tikzstarpoints}{
      -- (90+2*\i*\a-\a:\tikzstarradius/2) -- (90+2*\i*\a:\tikzstarradius)
    } -- cycle;
  }
}

\begin{tikzpicture}
 \draw[pattern=mystars,pattern color=blue]                (0,0) rectangle +(2,2);
 \draw[pattern={mystars[points=7,tile size=15pt]}]        (2,0) rectangle +(2,2);
 \draw[pattern={mystars[rotate=45]},pattern color=red]    (0,2) rectangle +(2,2);
 \draw[pattern={mystars[rotate=30,points=4,radius=5pt]}] (2,2) rectangle +(2,2);
\end{tikzpicture}
```

Instead of macros you can also use PGF keys as parameters, if that is what you prefer.



```
\usetikzlibrary {patterns.meta}
\tikzdeclarepattern{
  name=mylines,
  parameters={
      \pgfkeysvalueof{/pgf/pattern keys/size},
      \pgfkeysvalueof{/pgf/pattern keys/angle},
      \pgfkeysvalueof{/pgf/pattern keys/line width},
  },
  bounding box={
    (0,-0.5*\pgfkeysvalueof{/pgf/pattern keys/line width}) and
    (\pgfkeysvalueof{/pgf/pattern keys/size},
     0.5*\pgfkeysvalueof{/pgf/pattern keys/line width})},
  tile size={(\pgfkeysvalueof{/pgf/pattern keys/size},
              \pgfkeysvalueof{/pgf/pattern keys/size})},
  tile transformation={rotate=\pgfkeysvalueof{/pgf/pattern keys/angle}},
  defaults={
    size/.initial=5pt,
    angle/.initial=45,
    line width/.initial=.4pt,
  },
  code={
      \draw [line width=\pgfkeysvalueof{/pgf/pattern keys/line width}]
        (0,0) -- (\pgfkeysvalueof{/pgf/pattern keys/size},0);
  },
}

\begin{tikzpicture}
  \draw[pattern={mylines[size=10pt,line width=.8pt,angle=10]},
        pattern color=red]    (0,0) rectangle ++(2,2);
  \draw[pattern={mylines[size= 5pt,line width=.8pt,angle=40]},
        pattern color=blue]   (2,0) rectangle ++(2,2);
  \draw[pattern={mylines[size=10pt,line width=.4pt,angle=90]},
        pattern color=green]  (0,2) rectangle ++(2,2);
  \draw[pattern={mylines[size= 2pt,line width= 1pt,angle=70]},
        pattern color=orange] (2,2) rectangle ++(2,2);
\end{tikzpicture}
```

63 Three Point Perspective Drawing Library

by Max Snippe

TikZ Library perspective

```
\usetikzlibrary{perspective} % LaTeX and plain TeX
\usetikzlibrary[perspective] % ConTeXt
```

This library provides tools for perspective drawing with one, two, or three vanishing points.

63.1 Coordinate Systems

Coordinate system three point perspective

The `three point perspective` coordinate system is very similar to the `xyz` coordinate system, save that it will display the provided coordinates with a perspective projection.

/tikz/cs/x=⟨*number*⟩ (no default, initially `0`)

The x component of the coordinate. Should be given *without* unit.

/tikz/cs/y=⟨*number*⟩ (no default, initially `0`)

Same as `x`.

/tikz/cs/z=⟨*number*⟩ (no default, initially `0`)

Same as `x`.

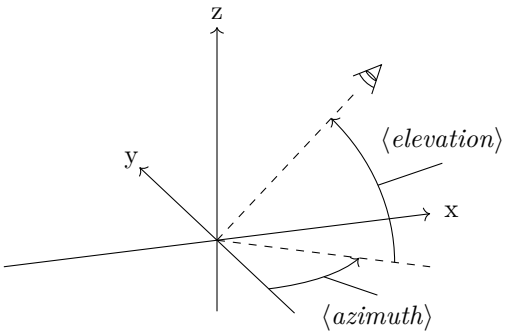
Coordinate system tpp

The `tpp` coordinate system is an alias for the `three point perspective` coordinate system.

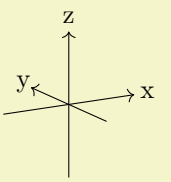
63.2 Setting the view

/tikz/3d view={⟨*azimuth*⟩}{⟨*elevation*⟩} (default `{-30}{15}`)

With the `3d view` option, the projection of the 3D coordinates on the 2D page is defined. It is determined by rotating the coordinate system by $-\langle azimuth\rangle$ around the z-axis, and by $\langle elevation\rangle$ around the (new) x-axis, as shown below.



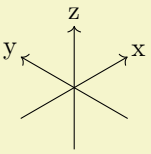
For example, when both $\langle azimuth\rangle$ and $\langle elevation\rangle$ are 0°, $+z$ will be pointing upward, and $+x$ will be pointing right. The default is as shown below.




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[3d view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```

/tikz/isometric view (style, no value)

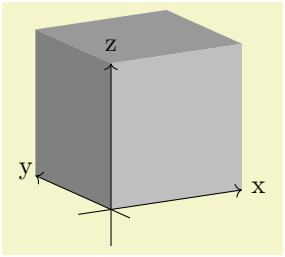
A special kind of `3d view` is isometric, which can be set with the `isometric view` style. It simply sets `3d view={-45}{35.26}`. The value for ⟨*elevation*⟩ is determined with $\arctan(1/\sqrt{2})$. In isometric projection the angle between any pair of axes is 120°, as shown below.




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[isometric view]
  \draw[->] (-1,0,0) -- (1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-1,0) -- (0,1,0) node[pos=1.1]{y};
  \draw[->] (0,0,-1) -- (0,0,1) node[pos=1.1]{z};
\end{tikzpicture}
```

63.3 Defining the perspective

In this section, the following example cuboid will be used with various scaling. As a reference, the axes will be shown too, without perspective projection.

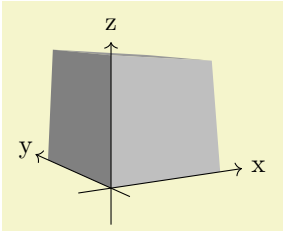



```
\usetikzlibrary {perspective}
\newcommand\simplecuboid[3]{%
  \fill[gray!80!white] (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=0,y=#2,z=#3)
    -- (tpp cs:x=#1,y=#2,z=#3)
    -- (tpp cs:x=#1,y=0,z=#3) -- cycle;
  \fill[gray]  (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=0,y=#2,z=#3)
    -- (tpp cs:x=0,y=#2,z=0) -- cycle;
  \fill[gray!50!white] (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=0,z=#3)
    -- (tpp cs:x=#1,y=0,z=#3)
    -- (tpp cs:x=#1,y=0,z=0) -- cycle;}
\newcommand{\simpleaxes}[3]{%
  \draw[->] (-0.5,0,0) -- (#1,0,0) node[pos=1.1]{x};
  \draw[->] (0,-0.5,0) -- (0,#2,0) node[pos=1.1]{y};
  \draw[->] (0,0,-0.5) -- (0,0,#3) node[pos=1.1]{z};}

\begin{tikzpicture}[3d view]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective=`⟨*vanishing points*⟩ (default `p={(10,0,0)},q={(0,10,0)},r={(0,0,20)}`)

The 'strength' of the perspective can be determined by setting the location of the vanishing points. The default values have a stronger perspective towards x and y than towards z, as shown below.

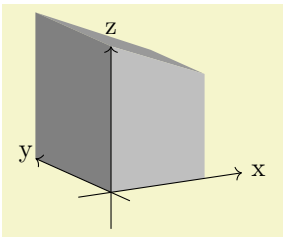



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[3d view,perspective]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

From this example it also shows that the maximum dimensions of the cuboid are no longer 2 by 2 by 2. This is inherent to the perspective projection.

`/tikz/perspective/p={`⟨x,y,z⟩`}` (no default, initially `(0,0,0)`)

The location of the vanishing point that determines the 'strength' of the perspective in x-direction can be set with the `p` key.

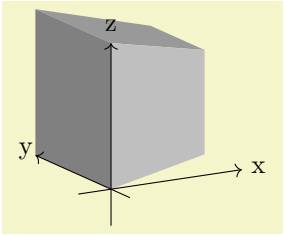



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    p = {(5,0,0)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

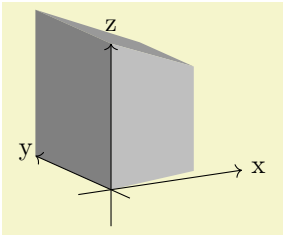
Note also that when only `p` is provided, the perspective in y and z direction is turned off.

To turn off the perspective in x-direction, one must set the x component of `p` to `0` (e.g. `p={(0,a,b)}`, where `a` and `b` can be any number and will be ignored). Or one can provide `q` and `r` and omit `p`.

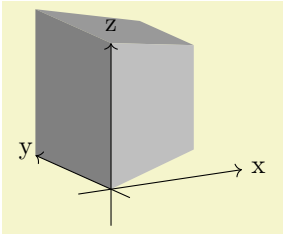
By changing the y and z components of `p`, one can achieve various effects.




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    p = {(5,0,1)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```



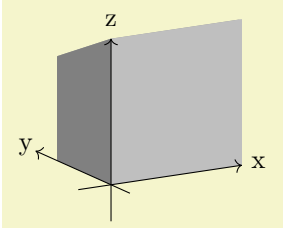

```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    p = {(5,1,0)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    p = {(5,1,1)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective/q={`$\langle x,y,z\rangle$`}` (no default, initially `(0,0,0)`)

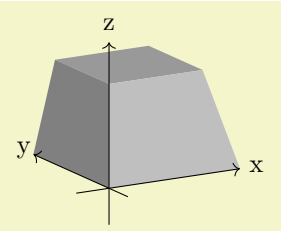
Similar to `p`, but can be turned off by setting its y component to `0`.




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    q = {(0,5,0)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

`/tikz/perspective/r={`$\langle x,y,z\rangle$`}` (no default, initially `(0,0,0)`)

Similar to `p`, but can be turned off by setting its z component to `0`.




```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    r = {(0,0,5)}}]
  \simplecuboid{2}{2}{2}
  \simpleaxes{2}{2}{2}
\end{tikzpicture}
```

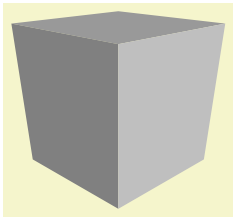
63.4 Shortcomings

Currently a number of things are not working, mostly due to the fact that PGF uses a 2D coordinate system underwater, and perspective projection is a non-linear affine transformation which needs to be aware of all three coordinates. These three coordinates are currently lost when processing a 3D coordinate. The issues include, but possibly are not limited to:

- Keys like `shift`, `xshift`, `yshift` are not working
- Keys like `rotate around x`, `rotate around y`, and `rotate around z` are not working
- Units are not working
- Most keys from the `3d` library are unsupported, e.g. all the `canvas is .. plane` keys.

63.5 Examples

An `r` that lies 'below' your drawing can mimic a macro effect.

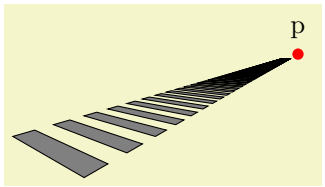


```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  isometric view,
  perspective={
    p = {(8,0,0)},
    q = {(0,8,0)},
    r = {(0,0,-8)}}]

  \simplecuboid{2}{2}{2}

\end{tikzpicture}
```

A peculiar phenomenon inherent to perspective drawing, is that however great your coordinate will become in the direction of the vanishing point, it will never reach it.



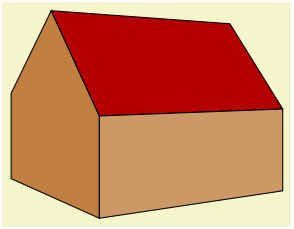

```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  isometric view,
  perspective={
    p = {(4,0,0)},
    q = {(0,4,0)}}]

    \node[fill=red,circle,inner sep=1.5pt,label=above:p] at (4,0,0){};

    \foreach \i in {0,...,100}{
      \filldraw[fill = gray] (tpp cs:x=\i,y=0,z=0)
        -- (tpp cs:x=\i+0.5,y=0,z=0)
        -- (tpp cs:x=\i+0.5,y=2,z=0)
        -- (tpp cs:x=\i,y=2,z=0)
        -- cycle;}
\end{tikzpicture}
```

Even for simple examples, the added perspective might add another 'dimension' to your drawing. In this

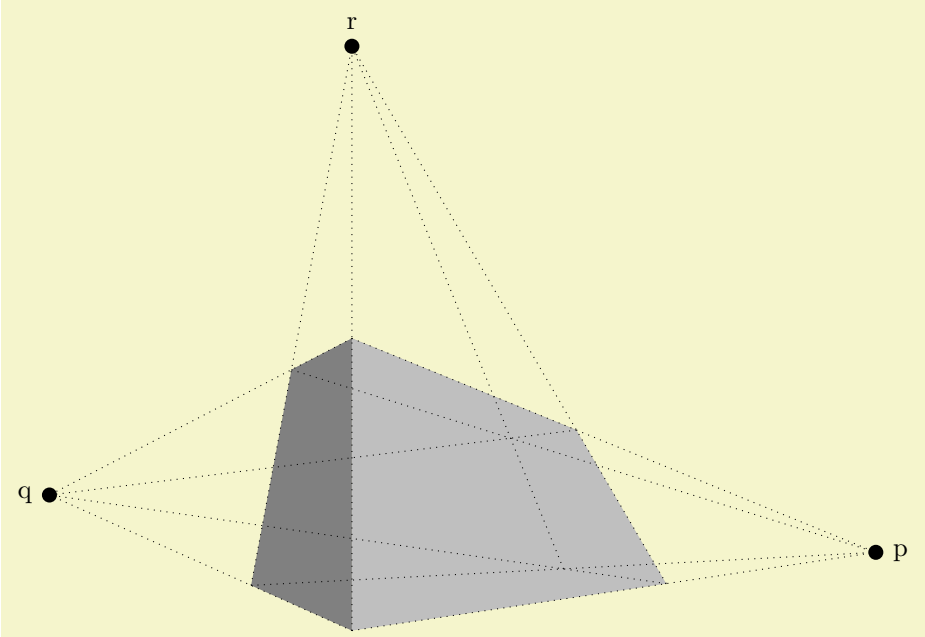
case, two vanishing points give a more intuitive result then three would.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  scale=0.7,
  3d view,
  perspective={
    p = {(20,0,0)},
    q = {(0,20,0)}}]

  \filldraw[fill=brown] (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=4,z=0)
    -- (tpp cs:x=0,y=4,z=2)
    -- (tpp cs:x=0,y=2,z=4)
    -- (tpp cs:x=0,y=0,z=2) -- cycle;
  \filldraw[fill=red!70!black] (tpp cs:x=0,y=0,z=2)
    -- (tpp cs:x=5,y=0,z=2)
    -- (tpp cs:x=5,y=2,z=4)
    -- (tpp cs:x=0,y=2,z=4) -- cycle;
  \filldraw[fill=brown!80!white] (tpp cs:x=0,y=0,z=0)
    -- (tpp cs:x=0,y=0,z=2)
    -- (tpp cs:x=5,y=0,z=2)
    -- (tpp cs:x=5,y=0,z=0) -- cycle;
\end{tikzpicture}
```

With the vanishing points nearby, the distortion of parallel lines becomes very strong. This might lead to `Dimension too large` errors.



```
\usetikzlibrary {perspective}
\begin{tikzpicture}[
  3d view,
  perspective={
    p = {(2,0,0)},
    q = {(0,2,0)},
    r = {(0,0,2)}},
  scale=4,
  vanishing point/.style={fill,circle,inner sep=2pt}]

  \simplecuboid{3}{1}{2}

  \node[vanishing point,label = right:p] (p) at (2,0,0){};
  \node[vanishing point,label = left:q] (q) at (0,2,0){};
  \node[vanishing point,label = above:r] (r) at (0,0,2){};

  \begin{scope}[dotted]
    \foreach \y in {0,1}{
      \foreach \z in {0,2}{
        \draw (tpp cs:x=0,y=\y,z=\z) -- (p.center);}}
    \foreach \x in {0,3}{
      \foreach \z in {0,2}{
        \draw (tpp cs:x=\x,y=0,z=\z) -- (q.center);}}
    \foreach \x in {0,3}{
      \foreach \y in {0,1}{
        \draw (tpp cs:x=\x,y=\y,z=0) -- (r.center);}}
  \end{scope}
\end{tikzpicture}
```

64 Petri-Net Drawing Library

TikZ Library petri

```
\usetikzlibrary{petri} % LaTeX and plain TeX
\usetikzlibrary[petri] % ConTeXt
```

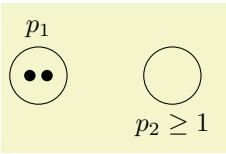
This packages provides shapes and styles for drawing Petri nets.

64.1 Places

The package defines a style for drawing places of Petri nets.

/tikz/place (style, no value)

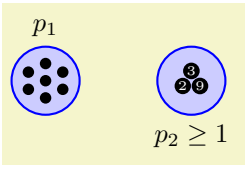
This style indicates that a node is a place of a Petri net. Usually, the text of the node should be empty since places do not contain any text. You should use the `label` option to add text outside the node like its name or its capacity. You should use the `tokens` options, explained in Section 64.3, to add tokens inside the place.




```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,label=above:$p_1$,tokens=2]         (p1) {};
  \node[place,label=below:$p_2\ge1$,right=of p1] (p2) {};
\end{tikzpicture}
```

/tikz/every place (style, no value)

This style is evoked by the style `place`. To change the appearance of places, you can change this style.




```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  [every place/.style={draw=blue,fill=blue!20,thick,minimum size=9mm}]
  \node[place,tokens=7,label=above:$p_1$]  (p1) {};
  \node[place,structured tokens={3,2,9},
        label=below:$p_2\ge1$,right=of p1] (p2) {};
\end{tikzpicture}
```

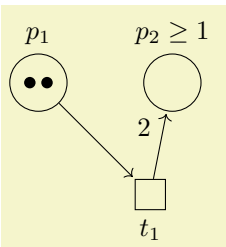
64.2 Transitions

Transitions are also nodes. They should be drawn using the following style:

/tikz/transition (style, no value)

This style indicates that a node is a transition. As for places, the text of a transition should be empty and the `label` option should be used for adding labels.

To connect a transition to places, you can use the `edge` command as in the following example:




```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,tokens=2,label=above:$p_1$]         (p1) {};
  \node[place,label=above:$p_2\ge1$,right=of p1] (p2) {};

  \node[transition,below right=of p1,label=below:$t_1$] {}
    edge[pre]                  (p1)
    edge[post] node[auto] {2} (p2);
\end{tikzpicture}
```

/tikz/every transition (style, no value)

This style is evoked by the style `transition`.

/tikz/pre (style, no value)

This style can be used with paths leading *from* a transition *to* a place to indicate that the place is in the pre-set of the transition. By default, this style is `<-,shorten <=1pt`, but feel free to redefine it.

/tikz/post (style, no value)

This style is also used with paths leading *from* a transition *to* a place, but this time the place is in the post-set of the transition. Again, feel free to redefine it.

/tikz/pre and post (style, no value)

This style is to be used to indicate that a place is both in the pre- and post-set of a transition.

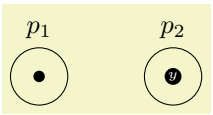
64.3 Tokens

Interestingly, the most complicated aspect of drawing Petri nets in Ti*k*Z turns out to be the placement of tokens.

Let us start with a single token. They are also nodes and there is a simple style for typesetting them.

/tikz/token (style, no value)

This style indicates that a node is a token. By default, this causes the node to be a small black circle. Unlike places and transitions, it *does* make sense to provide text for the token node. Such text will be typeset in a tiny font and in white on black (naturally, you can easily change this by setting the style `every token`).




```
\usetikzlibrary {petri,positioning}
\begin{tikzpicture}
  \node[place,label=above:$p_1$]              (p1) {};
  \node[token] at (p1) {};

  \node[place,label=above:$p_2$,right=of p1] (p2) {};
  \node[token] at (p2) {$y$};
\end{tikzpicture}
```

/tikz/every token (style, no value)

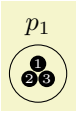
Change this style to change the appearance of tokens.

In the above example, it is bothersome that we need an extra command for the token node. Worse, when we have *two* tokens on a node, it is difficult to place both nodes inside the node without overlap.

The Petri library offers a solution to this problem: The `children are tokens` style.

/tikz/children are tokens (style, no value)

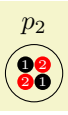
The idea behind this style is to use trees mechanism for placing tokens. Every token lying on a place is treated as a child of the node. Normally this would have the effect that the tokens are placed below the place and they would be connected to the place by an edge. The `children are tokens` style, however, redefines the growth function of trees such that it places the children next to each other inside (or, rather, on top) of the place node. Additionally, the edge from the parent node is not drawn.




```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node[place,label=above:$p_1$] {}
  [children are tokens]
  child {node [token] {1}}
  child {node [token] {2}}
  child {node [token] {3}};
\end{tikzpicture}
```

In detail, what happens is the following: Tree growth functions tell Ti*k*Z where it should place the children of nodes. These functions get passed the number of children that a node has an the number of the child that should be placed. The special tree growth function for tokens has a special mapping for each possible number of children up to nine children. This mapping decides for each child where it should be placed on top of the place. For example, a single child is placed directly on top of the

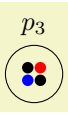
place. Two children are placed next to each other, separated by the `token distance`. Three children are placed in a triangle whose side lengths are `token distance`; and so on up to nine tokens. If you wish to place more than nice tokens on a place, you will have to write your own placement code.




```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node[place,label=above:$p_2$] {}
  [children are tokens]
  child {node [token] {1}}
  child {node [token,fill=red] {2}}
  child {node [token,fill=red] {2}}
  child {node [token] {1}};
\end{tikzpicture}
```

`/tikz/token distance`=⟨*distance*⟩ (no default)

This specifies the distance between the centers of the tokens in the arrangements of the option `children are tokens`.

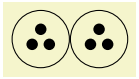



```
\usetikzlibrary {petri}
\begin{tikzpicture}
  \node[place,label=above:$p_3$] {}
  [children are tokens,token distance=1.1ex]
  child {node [token] {}}
  child {node [token,red] {}}
  child {node [token,blue] {}}
  child {node [token] {}};
\end{tikzpicture}
```

The `children are tokens` option gives you a lot of flexibility, but it is a bit cumbersome to use. For this reason there are some options that help in standard situations. They all use `children are tokens` internally, so any change to, say, the `every token` style will affect how these options depict tokens.

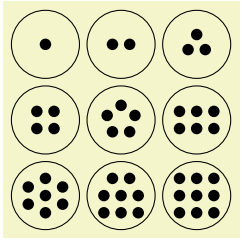
`/tikz/tokens`=⟨*number*⟩ (no default)

This option is given to a `place` node, not to a `token` node. The effect of this option is to add ⟨*number*⟩ many child nodes to the place, each having the style `token`. Thus, the following two pieces of codes have the same effect:



```
\usetikzlibrary {petri}
\tikz
  \node[place] {}
  [children are tokens]
  child {node [token] {}}
  child {node [token] {}}
  child {node [token] {}};
\tikz
  \node[place,tokens=3] {};
```

It is legal to say `tokens=0`, no tokens are drawn in this case. This option does not handle ten or more tokens correctly. If you need this many tokens, you will have to program your own code.



```
\usetikzlibrary {petri}
\begin{tikzpicture}[every place/.style={minimum size=9mm}]

  \foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                  0/1/4,1/1/5,2/1/6,
                                  0/0/7,1/0/8,2/0/9}
    \node [place,tokens=\tokennumber] at (\x,\y) {};
\end{tikzpicture}
```

`/tikz/colored tokens`=⟨*color list*⟩ (no default)

This option, which must also be given when a place node is being created, gets a list of colors as parameter. It will then add as many tokens to the place as there are colors in this list, each filled correspondingly.



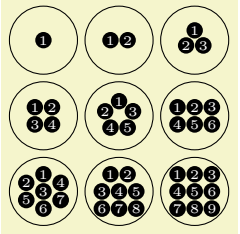
```
\usetikzlibrary {petri}
\tikz  \node[place,colored tokens={black,black,red,blue}] {};
```

`/tikz/structured tokens`=⟨*token texts*⟩ (no default)

This option, which must again be passed to a place, gets a list of texts for tokens. For each text, a new token will be added to the place.



```
\usetikzlibrary {petri}
\tikz  \node[place,structured tokens={$x$,$y$,$z$}] {};
```

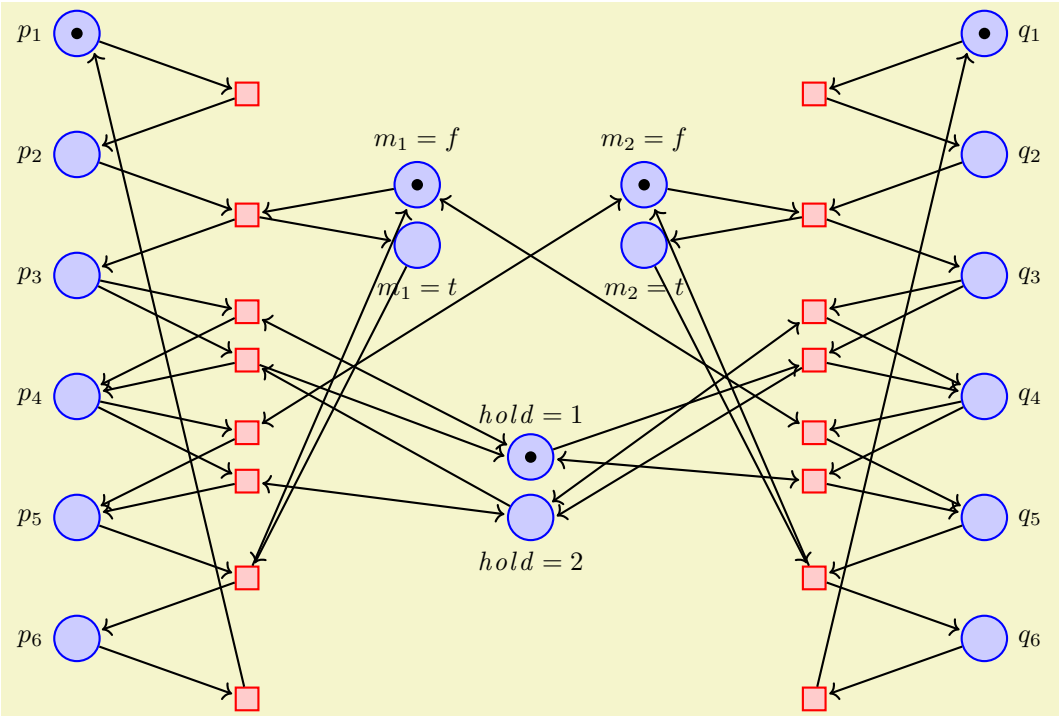


```
\usetikzlibrary {petri}
\begin{tikzpicture}[every place/.style={minimum size=9mm}]

  \foreach \x/\y/\tokennumber in {0/2/1,1/2/2,2/2/3,
                                  0/1/4,1/1/5,2/1/6,
                                  0/0/7,1/0/8,2/0/9}
    \node [place,structured tokens={1,...,\tokennumber}] at (\x,\y) {};
\end{tikzpicture}
```

If you use lots of structured tokens, consider redefining the `every token` style so that the tokens are larger.

64.4 Examples



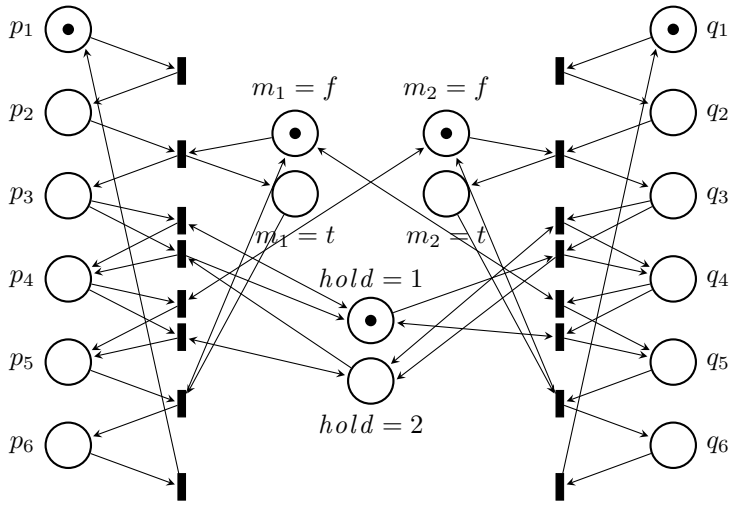
```
\usetikzlibrary {petri}
\begin{tikzpicture}[yscale=-1.6,xscale=1.5,thick,
  every transition/.style={draw=red,fill=red!20,minimum size=3mm},
  every place/.style={draw=blue,fill=blue!20,minimum size=6mm}]

  \foreach \i in {1,...,6} {
    \node[place,label=left:$p_\i$] (p\i) at (0,\i) {};
    \node[place,label=right:$q_\i$] (q\i) at (8,\i) {};
  }
  \foreach \name/\var/\vala/\valb/\height/\x in
      {m1/m_1/f/t/2.25/3,m2/m_2/f/t/2.25/5,h/\mathit{hold}/1/2/4.5/4} {
    \node[place,label=above:{$\var = \vala$}] (\name\vala) at (\x,\height) {};
    \node[place,yshift=-8mm,label=below:{$\var = \valb$}] (\name\valb) at (\x,\height) {};
  }
  \node[token] at (p1) {};   \node[token] at (q1) {};
  \node[token] at (m1f) {};  \node[token] at (m2f) {};
  \node[token] at (h1) {};

  \node[transition] at (1.5,1.5) {}  edge [pre] (p1)  edge [post] (p2);
  \node[transition] at (1.5,2.5) {}  edge [pre] (p2)  edge[pre]   (m1f)
                                     edge [post](p3)  edge[post]  (m1t);
  \node[transition] at (1.5,3.3) {}  edge [pre] (p3)  edge [post] (p4)
                                     edge [pre and post] (h1);
  \node[transition] at (1.5,3.7) {}  edge [pre] (p3)  edge [pre] (h2)
                                     edge [post] (p4) edge [post] (h1.west);
  \node[transition] at (1.5,4.3) {}  edge [pre] (p4)  edge [post] (p5)
                                     edge [pre and post] (m2f);
  \node[transition] at (1.5,4.7) {}  edge [pre] (p4)  edge [post] (p5)
                                     edge [pre and post] (h2);
  \node[transition] at (1.5,5.5) {}  edge [pre] (p5)  edge [pre] (m1t)
                                     edge [post] (p6) edge [post] (m1f);
  \node[transition] at (1.5,6.5) {}  edge [pre] (p6)  edge [post] (p1.south east);
  \node[transition] at (6.5,1.5) {}  edge [pre] (q1)  edge [post] (q2);
  \node[transition] at (6.5,2.5) {}  edge [pre] (q2)  edge [pre] (m2f)
                                     edge [post] (q3) edge [post] (m2t);
  \node[transition] at (6.5,3.3) {}  edge [pre] (q3)  edge [post] (q4)
                                     edge [pre and post] (h2);
  \node[transition] at (6.5,3.7) {}  edge [pre] (q3)  edge [pre] (h1)
                                     edge [post] (q4) edge [post] (h2.east);
  \node[transition] at (6.5,4.3) {}  edge [pre] (q4)  edge [post] (q5)
                                     edge [pre and post] (m1f);
  \node[transition] at (6.5,4.7) {}  edge [pre] (q4)  edge [post] (q5)
                                     edge [pre and post] (h1);
  \node[transition] at (6.5,5.5) {}  edge [pre] (q5)  edge [pre] (m2t)
                                     edge [post] (q6) edge [post] (m2f);
  \node[transition] at (6.5,6.5) {}  edge [pre] (q6)  edge [post] (q1.south west);
\end{tikzpicture}
```

Here is the same net once more, but with these styles changes:

```
\begin{tikzpicture}[yscale=-1.1,thin,>=stealth,
  every transition/.style={fill,minimum width=1mm,minimum height=3.5mm},
  every place/.style={draw,thick,minimum size=6mm}]
```



65 Plot Handler Library

Ti*k*Z Library plothandlers

```
\usepgflibrary{plothandlers}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[plothandlers]  % ConTeXt when using pure PGF
\usetikzlibrary{plothandlers} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plothandlers] % ConTeXt when using TikZ
```

This library packages defines additional plot handlers, see Section 112.3 for an introduction to plot handlers. The additional handlers are described in the following.

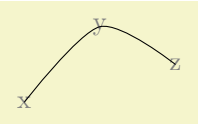
This library is loaded automatically by Ti*k*Z.

65.1 Curve Plot Handlers

\pgfplothandlercurveto

This handler will issue a \pgfpathcurveto command for each point of the plot, *except* possibly for the first. As for the line-to handler, what happens with the first point can be specified using \pgfsetmovetofirstplotpoint or \pgfsetlinetofirstplotpoint.

Obviously, the \pgfpathcurveto command needs, in addition to the points on the path, some control points. These are generated automatically using a somewhat "dumb" algorithm: Suppose you have three points x, y, and z on the curve such that y is between x and z:

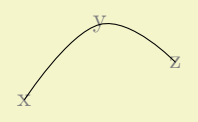



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

In order to determine the control points of the curve at the point y, the handler computes the vector $z - x$ and scales it by the tension factor (see below). Let us call the resulting vector s. Then $y + s$ and $y - s$ will be the control points around y. The first control point at the beginning of the curve will be the beginning itself, once more; likewise the last control point is the end itself.

\pgfsetplottension{⟨*value*⟩}

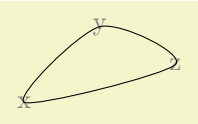
Sets the factor used by the curve plot handlers to determine the distance of the control points from the points they control. The higher the curvature of the curve points, the higher this value should be. A value of 1 will cause four points at quarter positions of a circle to be connected using a circle. The default is 0.5.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplottension{0.75}
  \pgfplothandlercurveto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerclosedcurve

This handler works like the curve-to plot handler, only it will add a new part to the current path that is a closed curve through the plot points.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerclosedcurve
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlermonotone

This handler will issue a \pgfpathcurveto command for each point of the plot, *except* possibly for the first. As for the line-to handler, what happens with the first point can be specified using \pgfsetmovetofirstplotpoint or \pgfsetlinetofirstplotpoint.

Obviously, the \pgfpathcurveto command needs, in addition to the points on the path, some control points. These are computed using The Fritsch-Carlson monotone cubic interpolation algorithm which roughly does the following:

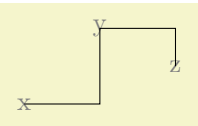
1. Initialize tangent slopes at each input point as the average of the slopes of the neighboring secant lines.
2. If the secants have opposite signes (because this is a local extremum), set the tangent slope to 0 to get an horizontal tangent and prevent overshoot. Same if one of the neighboring secants is horizontal.
3. Adjust the tangent slopes to ensure strict monotonicity.
4. Place the control points on the computed tangent lines, at one third of the x distance of a secant line: if a, b and c are three consecutive input points and m is the slope computed at B, then the control points around b will be u and v where $x_u = x_b - \frac{1}{3}(x_b - x_a)$, $y_u = y_b - \frac{1}{3}m(x_b - x_a)$, $x_v = x_b + \frac{1}{3}(x_c - x_b)$ and $y_v = y_b + \frac{1}{3}m(x_c - x_b)$.

65.2 Constant Plot Handlers

There are several plot handlers which produce piecewise constant interpolations between successive points:

\pgfplothandlerconstantlineto

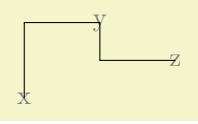
This handler works like the line-to plot handler, only it will produce a connected, piecewise constant path to connect the points.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerconstantlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerconstantlinetomarkright

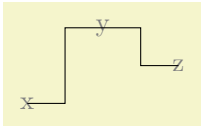
A variant of \pgfplothandlerconstantlineto which places its mark on the right line ends.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerconstantlinetomarkright
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerconstantlinetomarkmid

A variant of \pgfplothandlerconstantlineto which places its mark on the center of the line.

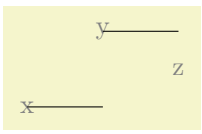



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerconstantlinetomarkmid
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

The plot handler always connects two data points by a horizontal line starting from the previous data points, followed by a vertical line in the middle between the two data points, followed by a horizontal line between the middle and the current data point. This results in a symmetric constant plot handler for constant mesh width.

\pgfplothandlerjumpmarkleft

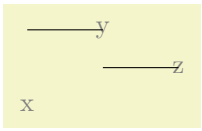
This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the left open pieces.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerjumpmarkleft
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerjumpmarkright

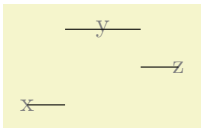
This handler works like the line-to plot handler, only it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed on the right open pieces.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerjumpmarkright
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerjumpmarkmid

This handler works like the \pgfplothandlerconstantlinetomarkmid, but it will produce a non-connected, piecewise constant path to connect the points. If there are any plot marks, they will be placed in the center of the horizontal line segment..




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerjumpmarkmid
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

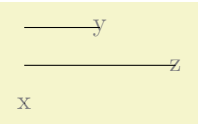
See \pgfplothandlerconstantlinetomarkmid for details.

65.3 Comb Plot Handlers

There are three "comb" plot handlers. Their name stems from the fact that the plots they produce look like "combs" (more or less).

\pgfplothandlerxcomb

This handler converts each point in the plot stream into a line from the y-axis to the point's coordinate, resulting in a "horizontal comb".

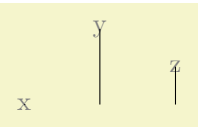



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerxcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerycomb

This handler converts each point in the plot stream into a line from the x-axis to the point's coordinate, resulting in a "vertical comb".

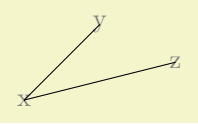
This handler is useful for creating "bar diagrams".




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerycomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerpolarcomb

This handler converts each point in the plot stream into a line from the origin to the point's coordinate.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerpolarcomb
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

PGF bar or comb plots usually draw something from zero to the current plot's coordinate. The "zero" offset can be changed using an input stream which returns the desired offset successively for each processed coordinate.

There are two such streams, which can be configured independently. The first one returns "zeros" for coordinate x, the second one returns "zeros" for coordinate y. They are used as follows.

```
\pgfplotxzerolevelstreamstart
\pgfplotxzerolevelstreamnext % assigns \pgf@x
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamnext
\pgfplotxzerolevelstreamend
```

```
\pgfplotyzerolevelstreamstart
\pgfplotyzerolevelstreamnext % assigns \pgf@x
\pgfplotyzerolevelstreamend
```

Different zero level streams can be implemented by overwriting these macros.

\pgfplotxzerolevelstreamconstant{⟨*dimension*⟩}

This zero level stream always returns {⟨*dimension*⟩} instead of $x = 0$pt.

It is used for `xcomb` and `xbar`.

\pgfplotyzerolevelstreamconstant{⟨*dimension*⟩}

This zero level stream always returns {⟨*dimension*⟩} instead of $y = 0$pt.

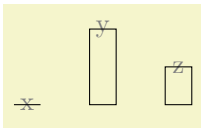
It is used for `ycomb` and `ybar`.

65.4 Bar Plot Handlers

While comb plot handlers produce a line-to operation to generate combs, bar plot handlers employ rectangular shapes, allowing filled bars (or pattern bars).

\pgfplothandlerybar

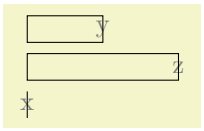
This handler converts each point in the plot stream into a rectangle from the x-axis to the point's coordinate. The rectangle is placed centered at the x-axis.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerybar
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlerxbar

This handler converts each point in the plot stream into a rectangle from the y-axis to the point's coordinate. The rectangle is placed centered at the y-axis.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlerxbar
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

/pgf/bar width={⟨*dimension*⟩} (no default, initially `10pt`)

alias `/tikz/bar width`

Sets the width of `\pgfplothandlerxbar` and `\pgfplothandlerybar` to {⟨*dimension*⟩}. The argument {⟨*dimension*⟩} will be evaluated using the math parser.

/pgf/bar shift={⟨*dimension*⟩} (no default, initially `0pt`)

alias `/tikz/bar shift`

Sets a shift used by `\pgfplothandlerxbar` and `\pgfplothandlerybar` to {⟨*dimension*⟩}. It has the same effect as `xshift`, but it applies only to those bar plots. The argument {⟨*dimension*⟩} will be evaluated using the math parser.

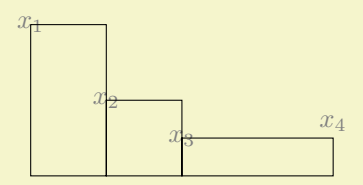
\pgfplotbarwidth

Expands to the value of `/pgf/bar width`.

\pgfplothandlerybarinterval

This handler is a variant of `\pgfplothandlerybar` which works with intervals instead of points.

Bars are drawn between successive input coordinates and the width is determined relatively to the interval length.




```
\begin{tikzpicture}
  \draw[gray] (0,2) node {$x_1$} (1,1) node {$x_2$} (2,.5) node {$x_3$} (4,0.7) node {$x_4$};
  \pgfplothandlerybarinterval
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

In more detail, if (x_i, y_i) and (x_{i+1}, y_{i+1}) denote successive input coordinates, the bar will be placed above the interval $[x_i, x_{i+1}]$, centered at

$$x_i + \langle \textit{bar interval shift} \rangle \cdot (x_{i+1} - x_i)$$

with width

$$\langle \textit{bar interval width} \rangle \cdot (x_{i+1} - x_i).$$

Here, $\langle \textit{bar interval shift} \rangle$ and $\langle \textit{bar interval width} \rangle$ denote the current values of the associated options.

If you have $N+1$ input points, you will get N bars (one for each interval). The y value of the last point will be ignored.

\pgfplothandlerxbarinterval

As `\pgfplothandlerybarinterval`, this handler provides bar plots with relative bar sizes and offsets, one bar for each y coordinate interval.

/pgf/bar interval shift={⟨*factor*⟩} (no default, initially `0.5`)

alias **/tikz/bar interval shift**

Sets the *relative* shift of `\pgfplothandlerxbarinterval` and `\pgfplothandlerybarinterval` to ⟨*factor*⟩. As **/pgf/bar interval width**, the argument is relative to the interval length of the input coordinates.

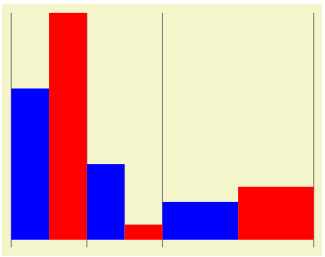
The argument {⟨*scale*⟩} will be evaluated using the math parser.

/pgf/bar interval width={⟨*scale*⟩} (no default, initially `1`)

alias **/tikz/bar interval width**

Sets the *relative* width of `\pgfplothandlerxbarinterval` and `\pgfplothandlerybarinterval` to {⟨*scale*⟩}. The argument is relative to $(x_{i+1} - x_i)$ for y bar plots and relative to $(y_{i+1} - y_i)$ for x bar plots.

The argument {⟨*scale*⟩} will be evaluated using the math parser.



```
\begin{tikzpicture}[bar interval width=0.5]
  \draw[gray]
    (0,3) -- (0,-0.1)
    (1,3) -- (1,-0.1)
    (2,3) -- (2,-0.1)
    (4,3) -- (4,-0.1);
  \pgfplothandlerybarinterval
  \begin{scope}[bar interval shift=0.25,fill=blue]
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{0.7cm}}
  \pgfplotstreamend
  \pgfusepath{fill}
  \end{scope}
  \begin{scope}[bar interval shift=0.75,fill=red]
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{3cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{0.2cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.7cm}}
  \pgfplotstreampoint{\pgfpoint{4cm}{0.2cm}}
  \pgfplotstreamend
  \pgfusepath{fill}
  \end{scope}
\end{tikzpicture}
```

Please note that bars are always centered, so we have to use shifts 0.25 and 0.75 instead of 0 and 0.5.

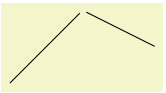
65.5 Gapped Plot Handlers

\pgfplothandlergaplineto

This handler will connect the points of the plots by straight line segments. However, at the start and the end of the lines there will be a small gap, given by the following key:

/pgf/gap around stream point=⟨*dimension*⟩ (no default, initially `1.5pt`)

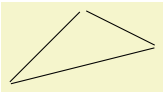
The ⟨*dimension*⟩ by which the lines between consecutive stream points are shortened at the beginning and end.



```
\begin{tikzpicture}
  \pgfplothandlergaplineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplothandlergapcycle

Works like \pgfplothandlergaplineto, but the last point is connected to the first in the same fashion:



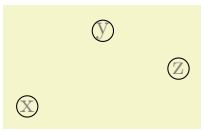
```
\begin{tikzpicture}
  \pgfplothandlergapcycle
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

65.6 Mark Plot Handler

\pgfplothandlermark{⟨*mark code*⟩}

This command will execute the ⟨*mark code*⟩ for some points of the plot, but each time the coordinate transformation matrix will be set up such that the origin is at the position of the point to be plotted. This way, if the ⟨*mark code*⟩ draws a little circle around the origin, little circles will be drawn at some point of the plot.

By default, a mark is drawn at all points of the plot. However, two parameters r and p influence this. First, only every rth mark is drawn. Second, the first mark drawn is the pth. These parameters can be influenced using the commands below.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

Typically, the ⟨*code*⟩ will be `\pgfuseplotmark{`⟨*plot mark name*⟩`}`, where ⟨*plot mark name*⟩ is the name of a predefined plot mark.

`\pgfsetplotmarkrepeat{`⟨*repeat*⟩`}`

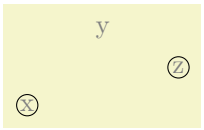
Sets the r parameter to ⟨*repeat*⟩, that is, only every rth mark will be drawn.

`\pgfsetplotmarkphase{`⟨*phase*⟩`}`

Sets the p parameter to ⟨*phase*⟩, that is, the first mark to be drawn is the pth, followed by the $(p+r)$th, then the $(p+2r)$th, and so on.

`\pgfplothandlermarklisted{`⟨*mark code*⟩`}{`⟨*index list*⟩`}`

This command works similar to the previous one. However, marks will only be placed at those indices in the given ⟨*index list*⟩. The syntax for the list is the same as for the `\foreach` statement. For example, if you provide the list `1,3,...,25`, a mark will be placed only at every second point. Similarly, `1,2,4,8,16,32` yields marks only at those points that are powers of two.




```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermarklisted
    {\pgfpathcircle{\pgfpointorigin}{4pt}\pgfusepath{stroke}}
    {1,3}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfuseplotmark{`⟨*plot mark name*⟩`}`

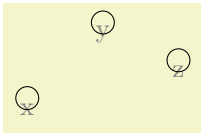
Draws the given ⟨*plot mark name*⟩ at the origin. The ⟨*plot mark name*⟩ must have been previously declared using `\pgfdeclareplotmark`.



```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{pentagon}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfdeclareplotmark{⟨*plot mark name*⟩}{⟨*code*⟩}

Declares a plot mark for later used with the \pgfuseplotmark command.

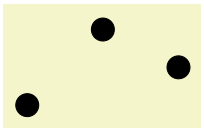


```
\pgfdeclareplotmark{my plot mark}
  {\pgfpathcircle{\pgfpoint{0cm}{1ex}}{1ex}\pgfusepathqstroke}
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfplothandlermark{\pgfuseplotmark{my plot mark}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetplotmarksize{⟨*dimension*⟩}

This command sets the TEX dimension \pgfplotmarksize to ⟨*dimension*⟩. This dimension is a "recommendation" for plot mark code at which size the plot mark should be drawn; plot mark code may choose to ignore this ⟨*dimension*⟩ altogether. For circles, ⟨*dimension*⟩ should be the radius, for other shapes it should be about half the width/height.

The predefined plot marks all take this dimension into account.



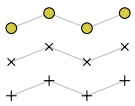
```
\begin{tikzpicture}
  \draw[gray] (0,0) node {x} (1,1) node {y} (2,.5) node {z};
  \pgfsetplotmarksize{1ex}
  \pgfplothandlermark{\pgfuseplotmark{*}}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{0.5cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfplotmarksize

A TEX dimension that is a "recommendation" for the size of plot marks.

The following plot marks are predefined (the filling color has been set to yellow):

\pgfuseplotmark{*}
\pgfuseplotmark{x}
\pgfuseplotmark{+}



66 Plot Mark Library

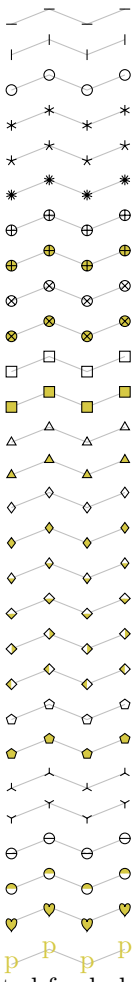
Ti*k*Z Library plotmarks

```
\usepgflibrary{plotmarks}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[plotmarks]  % ConTeXt when using pure PGF
\usetikzlibrary{plotmarks} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[plotmarks] % ConTeXt when using TikZ
```

This library defines a number of plot marks.

This library defines the following plot marks in addition to `*`, `x`, and `+` (the filling color has been set to a dark yellow):

```
\pgfuseplotmark{-}
\pgfuseplotmark{|}
\pgfuseplotmark{o}
\pgfuseplotmark{asterisk}
\pgfuseplotmark{star}
\pgfuseplotmark{10-pointed star}
\pgfuseplotmark{oplus}
\pgfuseplotmark{oplus*}
\pgfuseplotmark{otimes}
\pgfuseplotmark{otimes*}
\pgfuseplotmark{square}
\pgfuseplotmark{square*}
\pgfuseplotmark{triangle}
\pgfuseplotmark{triangle*}
\pgfuseplotmark{diamond}
\pgfuseplotmark{diamond*}
\pgfuseplotmark{halfdiamond*}
\pgfuseplotmark{halfsquare*}
\pgfuseplotmark{halfsquare right*}
\pgfuseplotmark{halfsquare left*}
\pgfuseplotmark{pentagon}
\pgfuseplotmark{pentagon*}
\pgfuseplotmark{Mercedes star}
\pgfuseplotmark{Mercedes star flipped}
\pgfuseplotmark{halfcircle}
\pgfuseplotmark{halfcircle*}
\pgfuseplotmark{heart}
\pgfuseplotmark{text}
```



Note that each of the provided marks can be rotated freely by means of `mark options={rotate=90}` or `every mark/.append style={rotate=90}`.

`/pgf/mark color=`{⟨*color*⟩} (no default, initially `empty`)

Defines the additional fill color for the `halfcircle`, `halfcircle*`, `halfdiamond*` and `halfsquare*` markers. An empty value uses `white` (which is the initial configuration). The special value `none` disables filling of the respective parts.

Note that `halfsquare` will be filled with `mark color`, and the starred variant `halfsquare*` will be filled half with `mark color` and half with the actual `fill` color.

`/pgf/text mark=`{⟨*text*⟩} (no default, initially `p`)

Changes the text shown by `mark=text`.

With `/pgf/text mark=m`: m m m m

With `/pgf/text mark=A`: A A A A

There is no limitation about the number of characters or whatever. In fact, any TeX material can be inserted as {⟨*text*⟩}, including images.

`/pgf/text mark as node={`⟨*boolean*⟩`}` (no default, initially `false`)

Configures how `mark=text` will be drawn: either as `\node` or as `\pgftext`.

The first choice is highly flexible and possibly slow, the second is very fast and usually enough.

`/pgf/text mark style={`⟨*options for* `mark=text`⟩`}` (no default)

Defines a set of options which control the appearance of `mark=text`.

If `/pgf/text mark as node=false` (the default), `{`⟨*options*⟩`}` is provided as argument to `\pgftext` – which provides only some basic keys like `left`, `right`, `top`, `bottom`, `base` and `rotate`.

If `/pgf/text mark as node=true`, `{`⟨*options*⟩`}` is provided as argument to `\node`. This means you can provide a very powerful set of options including `anchor`, `scale`, `fill`, `draw`, `rounded corners` etc.

67 Profiler Library

by Christian Feuersänger

Ti*k*Z Library profiler

```
\usepgflibrary{profiler}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[profiler]  % ConTeXt when using pure PGF
\usetikzlibrary{profiler} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[profiler] % ConTeXt when using TikZ
```

A library to simplify the optimization of runtime speed of TEX programs.

It relies on the `pdftex` primitive `\pdfelapsedtime`[13] to count (fractional) seconds and counts total time and self time for macro invocations.

67.1 Overview

The intended audience for this library are people writing TEX code which should be optimized. It is certainly *not* useful for the end-user.

The work flow for the optimization is simple: the preamble contains configuration commands like

```
\usepgflibrary{profiler}
\pgfprofilenewforenvironment{tikzpicture}
\pgfprofilenewforcommand{\pgfkeys}1}
```

and then, the time between `\begin{tikzpicture}` and `\end{tikzpicture}` and the time required to call `\pgfkeys` will be collected.

At the end, a short usage summary like

```
pgflibraryprofiler(main job) {total time=1.07378sec; (100.0122%) self time=0.034sec; (3.1662%)}
pgflibraryprofiler(<ENV>tikzpicture) {total time=1.03978sec; (96.84601%) self time=1.00415sec; (93.52722%)}
pgflibraryprofiler(<CS>pgfkeys) {total time=0.03563sec; (3.31726%) self time=0.03563sec; (3.31726%)}
```

will be provided in the log file, furthermore, the same information is available in a text table called `\jobname.profiler.`⟨*datetime*⟩`.dat` which is of the form:

profilerentry	totaltime[s]	totaltime[percent]	selftime[s]	selftime[percent]
main job	1.07378	100.0122	0.034	3.1662
<ENV>tikzpicture	1.03978	96.84601	1.00415	93.52722
<CS>pgfkeys	0.03563	3.31726	0.03563	3.31726

Here, the `totaltime` means the time used for all invocations of the respective profiler entry (one row in the table). The `selftime` measures time which is not already counted for in another profiler entry which has been invoked within the current one. The example above is not very exciting: the main job consists only of several (quite complex) pictures and nothing else. Thus, its total time is large. However, the self time is very small because the `tikzpicture`s are counted separately, and they have been invoked within the `main job`. The `\pgfkeys` control sequence has been invoked within the `tikzpicture`, that's why the `selftime` for the `tikzpicture` is a little bit smaller than its `totaltime`.

67.2 Requirements

The library works with `pdftex` and `luatex`. Furthermore, it requires a more or less recent version of `pdftex` which supports the `\pdfelapsedtime` directive.

67.3 Defining Profiler Entries

Unlike profilers for C/C++ or java, this library doesn't extract information about every TEX macro automatically, nor does it collect information for each of them. Instead, every profiler entry needs to be defined explicitly. Only defined profiler entries will be processed.

`\pgfprofilenew{`⟨*name*⟩`}`

Defines a new profiler entry named ⟨*name*⟩.

[13] The primitive is emulated in LuaTEX.

This updates a set of internal registers used to track the profiler entry. The ⟨*name*⟩ can be arbitrary, it doesn't need to be related to any TEX macro.

The actual job of counting seconds is accomplished using `\pgfprofilestart{`⟨*name*⟩`}` followed eventually by the command `\pgfprofileend{`⟨*name*⟩`}`.

It doesn't hurt if `\pgfprofilenew` is called multiple times with the same name.

`\pgfprofilenewforcommand[`⟨*profiler entry name*⟩`]{`⟨*\macro*⟩`}{`⟨*arguments*⟩`}`

Defines a new profiler entry which will measure the time spent in ⟨*\macro*⟩. This calls `\pgfprofilenew` and replaces the current definition of ⟨*\macro*⟩ with a new one.

If `[`⟨*profiler entry name*⟩`]` has been provided, this defines the argument for `\pgfprofilenew`. It is allowed to use the same name for multiple commands; in this case, they are treated as if it where the same command. If the optional argument is not used, the profiler entry will be called '`\pgfprofilecs`⟨*macro*⟩' (⟨*macro*⟩ without backslash) where `\pgfprofilecs` is predefined to be `<CS>`.

The replacement macro will collect all required arguments, start counting, invoke the original macro definition and stop counting.

The following macro types are supported within `\pgfprofilenewforcommand`:

- commands which take one (optional) argument in square brackets followed by one optional argument which has to be delimited by curly braces (use an empty argument for ⟨*arguments*⟩ in this case),
- commands which take one (optional) argument in square brackets and *exactly* ⟨*arguments*⟩ arguments afterwards.

Take a look at `\pgfprofilenewforcommandpattern` in case you have more complicated commands.

Note that the library can't detect if a command has been redefined somewhere.

`\pgfprofilenewforcommandpattern[`⟨*profiler entry name*⟩`]{`⟨*\macro*⟩`}{`⟨*argument pattern*⟩`}{`⟨*invocation pattern*⟩`}`

A variant of `\pgfprofilenewforcommand` which can be used with arbitrary ⟨*argument patterns*⟩. Example:

```
\def\mymacro#1\to#2\in#3{ ... }
\pgfprofilenewforcommandpattern{\mymacro}{#1\to#2\in#3}{{#1}\to{#2}\in{#3}}
```

Note that `\pgfprofilenewforcommand` is a special case of `\pgfprofilenewforcommandpattern`:

```
\def\mymacro#1#2{ ... }
\pgfprofilenewforcommand\macro{2}
\pgfprofilenewforcommandpattern{\mymacro}{#1#2}{{#1}{#2}}
```

Thus, ⟨*argument pattern*⟩ is a copy-paste from the definition of your command. The ⟨*invocation pattern*⟩ is used by the `profiler` library to invoke the *original* command, so it is closely related to ⟨*argument pattern*⟩, but it needs extra curly braces around each argument.

The behavior of `\pgfprofilenewforcommandpattern` is the same as discussed above: it defines a new profiler entry which will measure the time spent in ⟨*\macro*⟩. The details about this definition has already been described. Note that up to one optional argument in square brackets is also checked automatically.

If you like to profile a command which doesn't match here for whatever reasons, you'll have to redefine it manually and insert `\pgfprofilestart` and `\pgfprofileend` in appropriate places.

`\pgfprofileshowinvocationsfor{`⟨*profiler entry name*⟩`}`

Enables verbose output for *every* invocation of ⟨*profiler entry name*⟩.

This is only available for profiler entries for commands (those created by `\pgfprofilenewforcommand` for example). It will also show all given arguments.

`\pgfprofileshowinvocationsexpandedfor{`⟨*profiler entry name*⟩`}`

A variant of `\pgfprofileshowinvocationsfor` which will expand all arguments for ⟨*profiler entry name*⟩ before showing them. The invocation as such is not affected by this expansion.

This expansion (with `\edef`) might yield unrecoverable errors for some commands. Handle with care.

\pgfprofilenewforenvironment[⟨*profiler entry name*⟩]{⟨*environment name*⟩}

Defines a new profiler entry which measures time spent in the environment ⟨*environment name*⟩.

This calls \pgfprofilenew and handles the begin/end of the environment automatically.

The argument for \pgfprofilenew is ⟨*profiler entry name*⟩, or, if this optional argument is not used, it is '\pgfprofileenv⟨*environment name*⟩' where \pgfprofileenv is predefined as <ENV>. Again, it is permitted to use the same ⟨*profiler entry name*⟩ multiple times to merge different commands into one output section.

\pgfprofilestart{⟨*profiler entry name*⟩}

Starts (or resumes) timing of ⟨*profiler entry name*⟩. The argument must have been declared in the preamble using \pgfprofilenew.

Nested calls of \pgfprofilestart with the same argument will be ignored.

The invocation of this command doesn't change the environment: it doesn't introduce any TEX groups nor does it modify the token list.

\pgfprofileend{⟨*profiler entry name*⟩}

Stops (or interrupts) timing of ⟨*profiler entry name*⟩.

This command finishes a preceding call to \pgfprofilestart.

\pgfprofilepostprocess

For LATEX, this command is installed automatically in \end{document}. It stops all running timings, evaluates them and returns the result into the logfile. Furthermore, it generates a text table called \jobname.profiler.⟨*YYYY*⟩-⟨*MM*⟩-⟨*DD*⟩_⟨*HH*⟩h_⟨*MM*⟩m.dat with the same information.

Note that the profiler library predefines two profiler entries, namely main job which counts time from the beginning of the document until \pgfprofilepostprocess and preamble which counts time from the beginning of the document until \begin{document}.

\pgfprofilesetrel{⟨*profiler entry name*⟩} (initially main job)

Sets the profiler entry whose total time will be used to compute all other relative times. Thus, ⟨*profiler entry name*⟩ will use 100% of the total time per definition, all other relative times are relative to this one.

\pgfprofileifisrunning{⟨*profiler entry name*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

Invokes {⟨*true code*⟩} if {⟨*profiler entry name*⟩} is currently running and {⟨*false code*⟩} otherwise.

68 Resource Description Framework Library

With certain output formats (in particular, with SVG), Ti*k*Z can add *semantic annotations* to an output file. Consider as an example the drawing of a finite automaton. In your TEX code, you might have a nice description of the automaton like the following:

```
\tikz[automaton] \graph { a[state, initial] ->[transition] b [state] ->[transition] c[state, final] };
```

This description of the automaton carries a lot of "semantic information" like the information that the node `a` is not just some node, but actually the initial state of the automaton, while `c` is a final state. Unfortunately, in the output produced by Ti*k*Z, this information is normally "lost": In the output, `a` is only a short text, possibly with a circle drawn around it; but there is no information that *this* text and *this* circle together form the state of an automaton.

As a human (more precisely, as a computer scientist), you might "see" that the text and the circle form a state, but most software will have a very hard time retrieving this semantic information from the output. In particular, it is more or less impossible to design a search engine that you can query to find, say, "all automata with three states" in a document.

This is the point were *semantic annotations* come in. These are small labels or "hints" in the *output* that tell you (and, more importantly, a program) that the text and the circle together form a state of an automaton. There is a standard for specifying such annotations ("resource description framework annotations", abbreviated RDFa) and Ti*k*Z provides a way of adding such annotations to an output file using the `rdf engine` key, explained in a moment. Note, however, that the output format must support such annotations; currently Ti*k*Z only supports SVG.

68.1 Starting the RDF Engine

Ti*k*Z Library `rdf`

```
\usetikzlibrary{rdf} % LaTeX and plain TeX
\usetikzlibrary[rdf] % ConTeXt
```

You need to load this library for the keys described in the following. However, even when this library is loaded, RDF information is only written to the output inside scopes where the following key is set:

`/tikz/rdf engine on` (no value)

Switches "on" the generation of RDF information for the current TEX scope. The idea is that libraries can internally use the `rdf engine` key (explained below) a lot in order to provide good semantic information in the output when desired, but need not worry that this will bloat output files since users have to use this key explicitly to include semantic information in the output.

`/tikz/rdf engine=`⟨*rdf keys*⟩ (no default)

This key only has an effect when `rdf engine on` is called, otherwise the argument is silently ignored. The ⟨*rdf keys*⟩ get executed with the path prefix `/tikz/rdf engine` at the beginning of the current scope (for a node, at the beginning of the node's scope). Depending on which keys are used, semantic information gets to be added to the output.

Note that you cannot simply use the keys with path prefix `/tikz/rdf engine` directly since they need to be executed at very specific times during Ti*k*Z's processing of scopes. Always call those keys via this key.

The following key is useful for generally setting the prefix for a larger number of annotations:

`/tikz/rdf engine/prefix=`⟨*prefix: iri*⟩ (no default)

Inside the current scope, you can use ⟨*prefix*⟩`:` inside curies (compact universal resource identifier expressions, see the RDFA specification) as an abbreviation for the ⟨*iri*⟩. (It has the same effect as the `prefix` attribute in RDFa.) You can use this key several times for a given scope.

```
\scoped [rdf engine = {
  prefix = {rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns\tikzrdfhashmark},
  prefix = {automata: http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/},
  statement = { ..., predicate = rdf:type, object = automata:state },
  statement = { ..., predicate = rdf:type, object = automata:final },
  }] ...
```

The above could also be written more verbosely as

```
\scoped [rdf engine = {
  statement = { ...,
    predicate = http://www.w3.org/1999/02/22-rdf-syntax-ns\tikzrdfhashmark type,
    object = http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/state }
  },
  statement = { ...,
    predicate = http://www.w3.org/1999/02/22-rdf-syntax-ns\tikzrdfhashmark type,
    object = http://www.tcs.uni-luebeck.de/ontologies/2016/04/28/automata/final }
  }] ...
```

The use of the command `\tikzrdfhashmark` is necessary since TEX assigns a special meaning to hash marks. The command simple expands to a "normal" hash mark for use in texts.

`\tikzrdfhashmark`

Expands to # with catcode 11.

68.2 Creating Statements

Ti*k*Z's method of adding semantic information to an output is based on the principles underlying the *resource description framework* (RDF). In this framework, all semantic information is encoded using a large graph consisting of nodes and connecting directed edges, but the nodes are called *resources* and the edges are called *statements*. A resource is identified by an IRI, an *internationalized resource identifier*, which basically looks like the well-known URLs, but allows additional Unicode characters. Note that these IRIs do not need to point to "real" webpages, they are just a way of conceptually identifying resources uniquely and permanently. Similarly, each edge (statement) of the RDF graph has such an IRI attached to it, which identifies the "flavour" of the arc.

In a "mathematical" graph, each edge has a "tail" and a "head" vertex and a label, but in the context of the resource description framework these notions are called differently: As mentioned before, an edge is called a *statement*, the tail of this edge is called the *subject*, the head is called the *object* (in the linguistic sense), and the label is called the *predicate*. Thus, a statement is – quite fittingly – a triple "subject predicate object".

Note that in the RDF framework *all* semantic information must be encoded using statements of this fixed kind. Many semantic notions are easy to store in this way such as "Albert Einstein was a physicist" ("Albert Einstein" is the subject, "was" is the predicate, "a physicist" is the object), but other notions do not fit well like "The automaton has states q_1, q_2, q_a, and q_b" since there are several objects in the statement. Nevertheless, all information must be encoded as simple statements with a single subject, a single predicate, and a single object.

You add an RDF statement to the output file using the following key:

`/tikz/rdf engine/statement={`⟨*options*⟩`}` (no default)

Each use of this key will add one RDF statement to the output file. The ⟨*options*⟩ will be executed with the path prefix `/tikz/rdf engine/statements` and must use the three keys `subject`, `predicate`, and `object` to specify the three components of the statement (these keys can, however, be called by styles internally, so not all statements will explicitly set these three keys). Note that *all three must always be set*, it is *not* possible to setup, say, just a subject for a scope and then omit the subject for statements inside the scope. (However, using styles you can setup things in such a way that a certain subject is used for several statements.)

```
\tikz [rdf engine = {
  statement = {
    subject   = http://www.example.org/persons/Einstein,
    predicate = http://www.example.org/predicates/isA,
    object    = http://www.example.org/professions/physicist
  },
  statement = {
    subject   = http://www.example.org/persons/Curie,
    predicate = http://www.example.org/predicates/isA,
    object    = http://www.example.org/professions/physicist
  }}] { ... }
```

The statements are normally added at the beginning of the scope where the `rdf engine` command is used (except when the `object` is `scope content`, which is explained later). This means that when you

use `prefix` inside an `rdf engine` command, it will apply to all statements, regardless of the order.

`/tikz/rdf engine/statements/subject`=⟨*subject*⟩ (no default)

Sets the subject of the to-be-created statement. The ⟨*subject*⟩ can be in one of two possible formats:

1. A curie (a *compact universal resource identifier expression,* see the RDFA specification for details). Examples are standard URLs like `http://www.example.org`, but also text like `#my_automaton`. Note that in order to include a hashmark in a curie you should use the command `\tikzrdfhashmark`, which expands to a hash mark (TEX treats hash marks in a special way, which is why this command is used here).
2. When the ⟨*subject*⟩ starts with an opening parenthesis, that is, with "`(`", the ⟨*subject*⟩ must have the form `(`⟨*node or scope name*⟩`)`. In this case, the ⟨*node or scope name*⟩ must be the name of an already existing node (the current node or scope is considered as "existing" here). Then, the curie `#`⟨*id*⟩ is used as subject, where the ⟨*id*⟩ is a unique internal identifier for the node.
   As an example, suppose you wish to specify that a node has some other node as child, you could write the following:

```
\tikz [ rdf engine = { prefix = { rels: http://www.example.org/relations/} } ] {
  \node (fritz)          { Fritz };
  \node (heinz) at (2,0) { Heinz };
  \draw [->] (fritz) -- (heinz)
        [rdf engine = {
          statement = {
            subject = (fritz),
            predicate = rels:isSonOf,
            object  = (heinz)
          } } ];
}
```

You can use a macro as ⟨*subject*⟩, it will be expanded before the above syntax check is done.

If you use the `subject` key several times inside a single `statement` command, (only) the last subject is used.

`/tikz/rdf engine/statements/predicate`=⟨*predicate*⟩ (no default)

Sets the predicate for the statement. The syntax is exactly the same as for the subject. Unlike for subjects, you can use the predicate key several times inside a single statement and the uses will "accumulate" and several statements are created, namely one statement for each use of `predicate` for the subject and object specified inside the use of `statement`. This behavior is not very systematic (it violates the rule "one statement per `statement`") and you should normally use the `statement` once for each use of the `predicate` key. However, in conjunction with the object `scope content` it is necessary to allow this behavior.

`/tikz/rdf engine/statements/object`=⟨*object*⟩ (no default)

Sets the object for the statement. The syntax allowed for the ⟨*object*⟩ is as follows:

1. As for `subject` and `predicate` you can use a curie here. This is the default unless one of the following special cases is used:
2. As for `subject` and `predicate`, you can use the syntax `(`⟨*name of node or scope*⟩`)` to create and use a curie for the node or scope.
3. If the ⟨*object*⟩ starts with `"` and is of the form `"`⟨*literals*⟩`"`, the object of the statement is not a curie (not a normal "resource") but the string of ⟨*literals*⟩ given.
4. If the ⟨*object*⟩ is the text "`scope content`", the object of the statement is actually the whole contents of the scope to which this statement is attached.
5. Finally if the ⟨*object*⟩ is of the form `"`⟨*literals*⟩`" and scope content`, the contents of the scope is "normally" the object, but this gets "overruled" by the ⟨*literals*⟩. Formally, this means that the object is the ⟨*literals*⟩, but the intended semantics is that the object is the scope content, only for further processing it should be considered to be ⟨*literals*⟩. A typical example is the case where the scope content is, say, the text "January 1st, 2000" but the ⟨*literals*⟩ are set to `2000-01-01`, which is easier for software to process:

```
\node [rdf engine = {
  statement = {
    subject = ...,
    predicate = dc:Date,
    object = "2000-01-01" and scope content
  } } ] { January 1st, 2000 };
```

For the last two cases, only one statement may be given per scope that has the `scope content` as its object; if more than one is given, the last one wins. This is the reason why several uses of `predicate` are allowed in a `statement`.

`/tikz/rdf engine/statements/has type=`⟨*type*⟩ (no default)

This style is a shorthand for `predicate=rdf:type` and `object=`⟨*type*⟩.

68.3 Creating Resources

In RDF statements, when you can use the name of a Ti*k*Z scope or node surrounded by parenthesis, a curie is inserted into the output that references this scope or node. While this makes it easy to describe relationships between existing nodes, in library code (RDF generation code added to library styles and gets executed as a "byproduct") there are two situations where this method is insufficient:

1. You may sometimes wish to create additional resources in the RDF graph that are not represented by any concrete node or scope in the Ti*k*Z picture. For instance, a finite automaton has a set of states and a set of transitions, but neither of these "containers" has any concrete representation in the output. One could, of course, create "dummy scopes" for this purpose, but this is rather hard to do inside styles of a library.
2. You may not know the name of the current scope in library code. For instance, a style like `state`, which can be added to a node to indicate that the node is supposed to be a state in a finite automaton, might contain something like the following code:

```
state/.style = {
  draw, circle, minimum size = ...,
  rdf engine = {
    subject   = ???,
    predicate = rdf:type,
    object    = automata:state
  }
}
```

The problem is, of course, what should be passed to the `subject`. We cannot even write something like `(\tikz@fig@name)` since no name may have been set for the state.

Each of the above problems is solved by a special keys:

`/tikz/rdf engine/get new resource curie=`⟨*macro*⟩ (no default)

The ⟨*macro*⟩ will be set to a new unique curie that can be used anywhere where a curie is allowed. Here is an example how we can add a state and a transition container to an automaton, both of which have no corresponding scope in Ti*k*Z.

```
\tikz [ name = my automaton,
        rdf engine = {
          get new resource curie = \statecurie,
          get new resource curie = \transitioncurie,
          statement = {
            subject   = (my automaton),
            predicate = automata:hasStateSet,
            object    = \statecurie },
          statement = {
            subject   = \statecurie,
            hat type  = automata:stateSet },
          statement = {
            subject   = (my automaton),
            predicate = automata:hasTransitionSet,
            object    = \transitioncurie },
          statement = {
            subject   = \transitioncurie,
            hat type  = automata:transitionSet } } ] { ... }
```

The ⟨*macro*⟩ will be valid for the whole scope.

`/tikz/rdf engine/get scope curie=`⟨*macro*⟩ (no default)

The ⟨*macro*⟩ will be set to a unique curie that represents the scope or node. If the scope is named (using the `name` key or the special parenthesis syntax for nodes) and this name is later referenced in another statement, the same curie will be generated. Note how in the following code no name is given for the automaton, which means that the whole RDF code could be moved inside a style like `finite automaton` or something similar.

```
\tikz [ rdf engine = {
          get new resource curie = \statecurie,
          get new resource curie = \transitioncurie,
          get scope curie = \automatoncurie,
          statement = {
            subject   = \automatoncurie,
            predicate = automata:hasStateSet,
            object    = \statecurie },
          statement = {
            subject   = \statecurie,
            hat type  = automata:stateSet },
          statement = {
            subject   = \automatoncurie,
            predicate = automata:hasTransitionSet,
            object    = \transitioncurie },
          statement = {
            subject   = \transitioncurie,
            hat type  = automata:transitionSet } } ] { ... }
```

The ⟨*macro*⟩ will be valid for the whole scope.

The following key builds on the above keys:

`/tikz/rdf engine/scope is new context` (no value)

This key executes `get scope curie=\tikzrdfcontext`, thereby setting the macro `\tikzrdfcontext` to the current scope. The idea is the key is used with "major resources" and that keys can use this macro as the `subject` of statements if no subject is given explicitly. For instance, a `title` key might be defined as follows:

```
title/.style = {
  rdf engine = { statement = {
      subject   = \tikzrdfcontext,
      predicate = dc:Title,
      object    = "#1"
} } }
```

68.4 Creating Containers

A *container* is a resource that represents a set or a sequence of elements. In RDF this is modeled by having a statement say that the type of the resource is something special like `rdf:Seq` and then for each member resource of the container add a statement saying that the container has as its ith member the member resource. Here is an example of a container with two elements:

```
\tikz {
  \node (safe) { Safe }
  child { node (coins) {Coins} }
  child { node (gold)  {Gold} };

  \scoped [rdf engine = {
    statement = {
      subject   = (safe),
      has type  = rdf:Seq
    },
    statement = {
      subject   = (safe),
      predicate = rdf:_1,
      object    = (coins)
    },
    statement = {
      subject   = (safe),
      predicate = rdf:_2,
      object    = (gold)
    } }];
}
```

However, the above code is error-prone and does not integrate well with styles and libraries. For this reason, Ti*k*Z offers some styles that may help in creating containers:

`/tikz/rdf engine/statements/is a container` (no value)

Add this key to a statement in order to tell Ti*k*Z that it should setup a special counter for the subject of the statement that keeps track of the container's children.

`/tikz/rdf engine/statements/has as member` (no value)

This key may only be added to statements whose subject was previously used as a subject in a statement containing the `is a container` key. In this case, the internal counter will be increased and the predicate will be set to `rdf:_`⟨*count*⟩. This means that we can write the above code as:

```
\tikz { ...

  \scoped [rdf engine = {
    statement = {
      subject   = (safe),
      has type  = rdf:Seq,
      is a container,
    },
    statement = {
      subject   = (safe),
      has as member,
      object    = (coins)
    },
    statement = {
      subject   = (safe),
      has as member,
      object    = (gold)
    } }];
}
```

`/tikz/rdf engine/statements/is a sequence` (no value)

This is a shorthand for `predicate = rdf:Seq, is a container`. In the above example we could say:

```
\tikz { ...

  \scoped [rdf engine = {
    statement = {
      subject   = (safe),
      is a sequence
    },
    ... } ]; }
```

`/tikz/rdf engine/statements/is a bag` (no value)

This is a shorthand for `predicate = rdf:Bag, is a container`.

/tikz/rdf engine/statements/is an alternative (no value)

This is a shorthand for `predicate = rdf:Alt, is a container`.

68.5 Creating Semantic Information Inside Styles and Libraries

The RDF library was designed in such a way that normal document authors do not need to use the keys of the library explicitly, except possibly for saying `rdf engine on` somewhere at the beginning. Instead, library authors should include the necessary commands to generate RDF information that is then automatically included in the output. Furthermore, if the author does not "switch on" the generation of RDF information, all uses of `rdf engine` will simply be ignored silently and neither the speed of compilation nor the size of the generated files is impacted.

68.5.1 An Example Library for Drawing Finite Automata

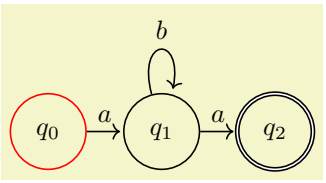
In the following, we have a look at how a library might be augmented by RDF generation keys. The library we augment is a (fictitious) library for drawing finite automata. The library offers the following styles:

1. `dfa` and `nfa` can be added to a scope to indicate that the scope contains a deterministic or a nondeterministic finite automaton.
2. `state` can be added to a node to indicate that the node is a state in the automaton.
3. `initial` and `final` are used to indicate that a state is an initial or final state, which should be rendered in a special way.
4. `transition` can be added to an edge to indicate that there is a transition in the automaton from the first state to the second state. The style takes a parameter which is the symbol read by the automaton.

Here are some possible definitions of these keys that do not (yet) generate RDF information:

```
\tikzset{
  dfa/.style        = { semithick, > = To [sep] },
  nfa/.style        = { semithick, > = To [sep] },
  state/.style      = { circle, draw, minimum size = 1cm },
  final/.style      = { double },
  initial/.style    = { draw = red }, % to keep things simple
  transition/.style = { edge label = {$#1$} } }
```

The library could be used as follows:




```
\usetikzlibrary {graphs,rdf}
\tikz [dfa]
  \graph [math nodes, grow right = 1.5cm] {
    q_0 [state, initial] -> [transition = a]
    q_1 [state]          -> [transition = b, loop above]
    q_1                  -> [transition = a]
    q_2 [state, final]
  };
```

68.5.2 Adding Semantic Information About the Automata as a Whole

Let us change the different keys so that they add RDF information to the output. For this, we first need an ontology that defines notions like "state" or "deterministic finite automaton". For the purposes of this example, we just assume that such an ontology exists at `http://www.tcs.uni-luebeck.de/ontologies/automata/`. The new definition of the `dfa` key might start as follows (we will extend these later on):

```
dfa/.style = {
  semithick, > = To [sep], % as before,
  rdf engine = {
    %
    % Setup prefix:
    prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
    %
    % Get the curie of the automaton and store it in a macro for later use:
    get scope curie = \mylibAutomatonCurie,
    %
    % Make a statement that the resource is, indeed, an automaton:
    statement = {
      subject     = \mylibAutomatonCurie,
      has type    = automata:types/automaton },
    %
    % Make a statement that the automaton is deterministic:
    statement = {
      subject     = \mylibAutomatonCurie,
      predicate   = automata:properties/deterministic,
      object      = "yes" } } }
```

The definition of the style `nfa` would be exactly the same as for `dfa`, except, of course, that the last statement would have `"no"` as object. Note that the original styles `dfa` and `nfa` has identical definitions since, indeed, there is no "visual" difference between the two. In contrast, the RDF information stores this information in the output.

68.5.3 Adding Semantic Information About the States

We next augment the styles for creating states and marking them as final or initial. We could do the following (note that we do not setup the prefix since this has been done by the surrounding `dfa` or `nfa` key):

```
state/.style = {
  circle, draw, minimum size = 1cm, % as before,
  rdf engine = {
    get scope curie = \mylibStateCurie,
    statement = {
      subject     = \mylibStateCurie,
      has type    = automata:types/state },
    statement = {
      subject     = \mylibStateCurie,
      predicate   = rdf:value,
      object      = scope content } } }
```

The first statement tells us that the circle with its contents is a state (and not just "any" fancy circle). The second statement tells us that the "value" of this state is the content. One might argue that, instead, only the number itself (like "q_0" or perhaps only "0") should be the "value" or, perhaps, a different property should actually be used (like `automata:stateNumber` or something like that). However, these are questions of ontological modeling, not of the use of the RDF engine.

What is definitely missing from the above definition is a link between the automaton resource and the state resource. Note that the state's rendering code will be inside the scope of the automaton, so, in a sense, the state is "inside" the automaton in the output. However, the nesting of scopes is *not* part of the RDF graph; we must make these relationships explicit using statements. One way to achieve this would be to add the following to the `state` style:

```
    statement = {
      subject     = \mylibAutomatonCurie,
      predicate   = automata:hasAsAState,
      object      = \mylibStateCurie }
```

Note that we can access the macro `\mylibAutomatonCurie` here since this will have been setup by the surrounding `dfa` or `nfa` key. While the above is possible and legitimate, we will see a better solution using containers in a moment ("better" in the sense that the ontological model is easier to process by software).

The two styles `final` and `initial` are easy to augment:

```
final/.style = {
  double, % as before,
  rdf engine = {
    get scope curie = \mylibStateCurie,
    statement = {
      subject     = \mylibStateCurie,
      has type    = automata:properties/final } } }
```

Note that when we write `node [state, final] ...` the state now has two types: It has type `automata:types/state` and also `automata:properties/final`. This is perfectly legitimate. Also note that I added `get scope curie` to the above definition, which may seem superfluous since the `state` style already executes this key to get a curie for the state resource. However, users should be free to write `node [final, state] ...` and, now, the `final` key will be executed first.

The style for `initial` is the same as for `final` only with a different type.

68.5.4 Adding Semantic Information About the Transitions

A transition is, essentially, a labeled edge from a state to another state. It may seem tempting to model them as a statement with the first state as its subject and the second state as the object and the transition's symbol as the label (turned into a curie in some appropriate way). However, closer inspection shows that this is not a good way of modeling transitions: In essence, it is just coincidence that the RDF graph happens to be a directed graph and, at the same time, the thing we describe by it (the automaton) can also be viewed as a directed graph. If, for instance, we consider alternating automata where a transition can involve more than two states, the simple model breaks down.

The "right" way of modeling a transition is to treat the transition as a resource of its own and then make statements like "the transition has this state as its old state". This turns out to be relatively easy to achieve:

```
transition/.style = {
  edge label = {#1}, % as before,
  rdf engine = {
    get scope curie = \mylibTransitionCurie,
    statement = {
      subject     = \mylibTransitionCurie,
      has type    = automata:types/transition },
    statement = {
      subject     = \mylibTransitionCurie,
      predicate   = automata:properties/symbolReadFromTape,
      object      = "#1" },
    statement = {
      subject     = \mylibTransitionCurie,
      predicate   = automata:relations/oldState,
      object      = (\tikztostart) },
    statement = {
      subject     = \mylibTransitionCurie,
      predicate   = automata:relations/newState,
      object      = (\tikztotarget) } } }
```

68.5.5 Using Containers

As a last step we wish to organize the states and transitions using containers. As explained earlier, we can easily add statements linking our automaton to the states and to the transitions, but the RDF standard has standard way of specifying that a set of resources form a logical sequence: containers.

In case automata contained *only* states, we could setup the automaton itself to be the container and the states to be its elements. However, the automaton has states and transitions and in this example I would like to keep these in separate containers. Thus, we must create two containers and then make statements that the automaton contains these two containers. Since these containers do not have any accompanying visual representation, we use the `get new resource curie` key to create new resources that are purely for descriptive purposes inside the RDF graph:

```
dfa/.style = {
  semithick, > = To [sep], % as before,
  rdf engine = {
    prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
    get scope curie = \mylibAutomatonCurie,
    statement = { ... as before that automaton has type automata:types/automaton ... },
    statement = { ... as before that automaton is deterministic ... },
    get new resource curie = \mylibStateContainerCurie,
    statement = {
      subject    = \mylibStateContainerCurie,
      is a sequence },
    statement = {
      subject    = \mylibAutomatonCurie,
      predicate  = automata:relations/hasAsStateContainer,
      object     = \mylibStateContainerCurie },
    get new resource curie = \mylibTransitionContainerCurie,
    statement = {
      subject    = \mylibTransitionContainerCurie,
      is a sequence },
    statement = {
      subject    = \mylibAutomatonCurie,
      predicate  = automata:relations/hasAsTransitionContainer,
      object     = \mylibTransitionContainerCurie } } }
```

We can now modify the `state` style as follows:

```
state/.style = {
  circle, draw, minimum size = 1cm, % as before,
  rdf engine = {
    get scope curie = \mylibStateCurie,
    statement = { ... as before ... },
    statement = { ... as before ... },
    statement = {
      subject     = \mylibStateContainerCurie,
      has as member,
      object      = \mylibStateCurie } } }
```

The modification for the `transition` style is similar:

```
transition/.style = {
  edge label = {#1}, % as before,
  rdf engine = {
    get scope curie = \mylibTransitionCurie,
    statement = { ... as before ... },
    statement = { ... as before ... },
    statement = { ... as before ... },
    statement = { ... as before ... },
    statement = {
      subject     = \mylibTransitionContainerCurie,
      has as member,
      object      = \mylibTransitionCurie } } }
```

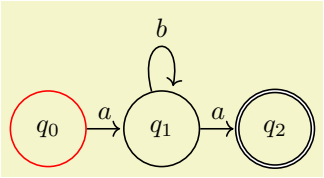
68.5.6 The Resulting RDF Graph

Putting it all together, we now get the following library code:

```
\tikzset{
  dfa/.style = {
    semithick, > = To [sep], % as before,
    rdf engine = {
      prefix = { automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ },
      get scope curie = \mylibAutomatonCurie,
      statement = {
        subject     = \mylibAutomatonCurie,
        has type    = automata:types/automaton },
      statement = {
        subject     = \mylibAutomatonCurie,
        predicate   = automata:properties/deterministic,
        object      = "yes" },
      get new resource curie = \mylibStateContainerCurie,
      statement = {
        subject    = \mylibStateContainerCurie,
        is a sequence },
```

```
      statement = {
        subject    = \mylibAutomatonCurie,
        predicate  = automata:relations/hasAsStateContainer,
        object     = \mylibStateContainerCurie },
      get new resource curie = \mylibTransitionContainerCurie,
      statement = {
        subject    = \mylibTransitionContainerCurie,
        is a sequence },
      statement = {
        subject    = \mylibAutomatonCurie,
        predicate  = automata:relations/hasAsTransitionContainer,
        object     = \mylibTransitionContainerCurie } } },
  state/.style = {
    circle, draw, minimum size = 1cm, % as before,
    rdf engine = {
      get scope curie = \mylibStateCurie,
      statement = {
        subject     = \mylibStateCurie,
        has type    = automata:types/state },
      statement = {
        subject     = \mylibStateCurie,
        predicate   = rdf:value,
        object      = scope content },
      statement = {
        subject     = \mylibStateContainerCurie,
        has as member,
        object      = \mylibStateCurie } } },
  initial/.style = {
    draw = red, % as before,
    rdf engine = {
      get scope curie = \mylibStateCurie,
      statement = {
        subject     = \mylibStateCurie,
        has type    = automata:properties/initial } } },
  final/.style = {
    double, % as before,
    rdf engine = {
      get scope curie = \mylibStateCurie,
      statement = {
        subject     = \mylibStateCurie,
        has type    = automata:properties/final } } },
  transition/.style = {
    edge label = {$#1$}, % as before,
    rdf engine = {
      get scope curie = \mylibTransitionCurie,
      statement = {
        subject     = \mylibTransitionCurie,
        has type    = automata:types/transition },
      statement = {
        subject     = \mylibTransitionCurie,
        predicate   = automata:properties/symbolReadFromTape,
        object      = "#1" },
      statement = {
        subject     = \mylibTransitionCurie,
        predicate   = automata:relations/oldState,
        object      = (\tikztostart) },
      statement = {
        subject     = \mylibTransitionCurie,
        predicate   = automata:relations/newState,
        object      = (\tikztotarget) },
    statement = {
      subject     = \mylibTransitionContainerCurie,
      has as member,
      object      = \mylibTransitionCurie } } }
}
```

Using this code is still "as easy as before", indeed, the code for creating the automaton is perfectly unchanged:



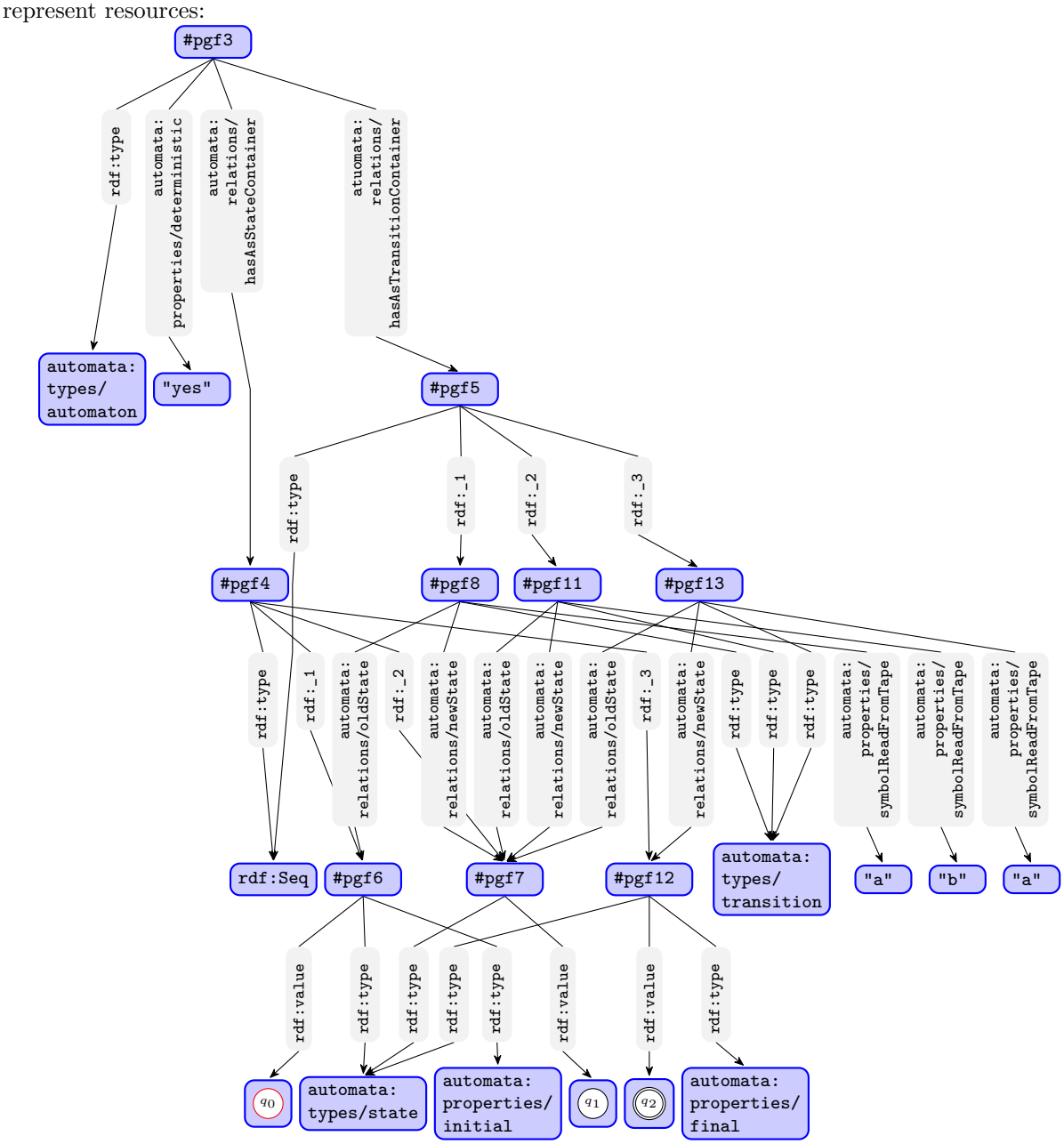

```
\usetikzlibrary {graphs,rdf}
\tikzset{  rdf engine on }
\tikz [dfa]
  \graph [math nodes, grow right = 1.5cm] {
    q_0 [state, initial] -> [transition = a]
    q_1 [state]          -> [transition = b, loop above]
    q_1                  -> [transition = a]
    q_2 [state, final]
  };
```

Let us now have a look at the result. If the above is processed using TEX and transformed to SVG code, the following results (reformatted and slightly simplified):

```
<g id="pgf3"  prefix=" automata: http://www.tcs.uni-luebeck.de/ontologies/automata/ ">
  <!-- The automaton -->
  <g about="#pgf3" property="rdf:type" resource="automata:types/automaton" />
  <g about="#pgf3" property="automata:properties/deterministic" content="yes" />
  <g about="#pgf4" property="rdf:type" resource="rdf:Seq" />
  <g about="#pgf3" property="automata:relations/hasAsStateContainer" resource="#pgf4" />
  <g about="#pgf5" property="rdf:type" resource="rdf:Seq" />
  <g about="#pgf3" property="automata:relations/hasAsTransitionContainer" resource="#pgf5" />
  <g id="pgf6"  about="#pgf6" property="rdf:value">
    <!-- State $q_0$ -->
    <g about="#pgf6" property="rdf:type" resource="automata:types/state" />
    <g about="#pgf4" property="rdf:_1" resource="#pgf6" />
    <g about="#pgf6" property="rdf:type" resource="automata:properties/initial" />
    <g stroke="#f00">    <!-- Red Line -->
      <path id="pgf6bp" d="M 14.22636 0.0 C 14.22636 7.8571 7.8571 14.22636 0.0 14.22636 ..." />
      ...
    </g>
  </g>
  <g id="pgf7"  about="#pgf7" property="rdf:value">
    <!-- State $q_1$ -->
    <g about="#pgf7" property="rdf:type" resource="automata:types/state" />
    <g about="#pgf4" property="rdf:_2" resource="#pgf7" />
    <path id="pgf7bp" d="M 56.90549 0.0 C 56.90549 7.8571 50.53622 14.22636 42.67912 14.22636 ..." />
    ...
  </g>
  <g id="pgf8" >
    <!-- Transition from $q_0$ to $q_1$ -->
    <g about="#pgf8" property="rdf:type" resource="automata:types/transition" />
    <g about="#pgf8" property="automata:properties/symbolReadFromTape" content="a" />
    <g about="#pgf8" property="automata:relations/oldState" resource="#pgf6" />
    <g about="#pgf8" property="automata:relations/newState" resource="#pgf7" />
    <g about="#pgf5" property="rdf:_1" resource="#pgf8" />
    <path id="pgf8p" d="M 14.52637 0.0 L 26.49275 0.0"/>
    ...
  </g>
  <g id="pgf11" >
    <!-- Transition loop at $q_1$ -->
    <g about="#pgf11" property="rdf:type" resource="automata:types/transition" />
    <g about="#pgf11" property="automata:properties/symbolReadFromTape" content="b" />
    <g about="#pgf11" property="automata:relations/oldState" resource="#pgf7" />
    <g about="#pgf11" property="automata:relations/newState" resource="#pgf7" />
    <g about="#pgf5" property="rdf:_2" resource="#pgf11" />
    <path id="pgf11p" d="M 38.91211 14.05888 C 33.07051 35.8591 51.00113 36.72765 47.05392 16.66444"/>
    ...
  </g>
  <g id="pgf12"  about="#pgf12" property="rdf:value">
    <!-- State $q_2$ -->
    <g about="#pgf12" property="rdf:type" resource="automata:types/state" />
    <g about="#pgf4" property="rdf:_3" resource="#pgf12" />
    <g about="#pgf12" property="rdf:type" resource="automata:properties/final" />
    <g stroke-width="1.80002">     <!-- Double Line -->
      <path id="pgf12bp" d="M 99.58461 0.0 C 99.58461 7.8571 93.21535 14.22636 85.35825 14.22636 ..." />
      ...
    </g>
  </g>
  <g id="pgf13" >
    <!-- Transition from $q_1$ to $q_2$ -->
    <g about="#pgf13" property="rdf:type" resource="automata:types/transition" />
    <g about="#pgf13" property="automata:properties/symbolReadFromTape" content="a" />
    <g about="#pgf13" property="automata:relations/oldState" resource="#pgf7" />
    <g about="#pgf13" property="automata:relations/newState" resource="#pgf12" />
```

```
    <g about="#pgf5" property="rdf:_3" resource="#pgf13" />
    <path id="pgf13p" d="M 57.20549 0.0 L 69.17188 0.0"/>
    ...
  </g>
</g>
```

When this code is processed by some RDFa tool, the following graph will result where the blue nodes represent resources:



69 Shadings Library

TikZ Library shadings

```
\usepgflibrary{shadings}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shadings]  % ConTeXt when using pure PGF
\usetikzlibrary{shadings} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shadings] % ConTeXt when using TikZ
```

The package defines a number of shadings in addition to the ball and axis shadings that are available by default.

In the following, the shadings defined in the library are listed in alphabetical order. The colors of some of these shadings can be configured using special options (like `left color`). These options implicitly select the shading.

The three shadings `axis`, `ball`, and `radial` are always defined, even when this library is not used.

Shading axis

In this always-defined shading the colors change gradually between three horizontal lines. The top line is at the top (uppermost) point of the path, the middle is in the middle, the bottom line is at the bottom of the path.

`/tikz/top color=`⟨*color*⟩ (no default)

This option sets the color to be used at the top in an `axis` shading. When this option is given, several things happen:

1. The `shade` option is selected.
2. The `shading=axis` option is selected.
3. The middle color of the axis shading is set to the average of the given top color ⟨*color*⟩ and of whatever color is currently selected for the bottom.
4. The rotation angle of the shading is set to 0.



```
\usepgflibrary {shadings}
\tikz \draw[top color=red] (0,0) rectangle (2,1);
```

`/tikz/bottom color=`⟨*color*⟩ (no default)

This option works like `top color`, only for the bottom color.

`/tikz/middle color=`⟨*color*⟩ (no default)

This option specifies the color for the middle of an axis shading. It also sets the `shade` and `shading=axis` options, but it does not change the rotation angle.

Note: Since both `top color` and `bottom color` change the middle color, this option should be given *last* if all of these options need to be given:



```
\usepgflibrary {shadings}
\tikz \draw[top color=white,bottom color=black,middle color=red]
  (0,0) rectangle (2,1);
```

`/tikz/left color=`⟨*color*⟩ (no default)

This option does exactly the same as `top color`, except that the shading angle is set to 90°.

`/tikz/right color=`⟨*color*⟩ (no default)

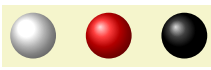
Works like `left color`.

Shading ball

This always-defined shading fills the path with a shading that “looks like a ball”. The default “color” of the ball is blue (for no particular reason).

/tikz/ball color=⟨*color*⟩ (no default)

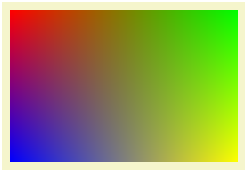
This option sets the color used for the ball shading. It sets the `shade` and `shading=ball` options. Note that the ball will never "completely" have the color ⟨*color*⟩. At its "highlight" spot a certain amount of white is mixed in, at the border a certain amount of black. Because of this, it also makes sense to say `ball color=white` or `ball color=black`



```
\usepgflibrary {shadings}
\begin{tikzpicture}
  \shade[ball color=white] (0,0) circle [radius=2ex];
  \shade[ball color=red]   (1,0) circle [radius=2ex];
  \shade[ball color=black] (2,0) circle [radius=2ex];
\end{tikzpicture}
```

Shading bilinear interpolation

This shading fills a rectangle with colors that a bilinearly interpolated between the colors in the four corners of the rectangle. These four colors are called `lower left`, `lower right`, `upper left`, and `upper right`. By changing these color, you can change the way the shading looks. The library also defines four options, called the same way, that can be used to set these colors and select the shading implicitly.



```
\usepgflibrary {shadings}
\tikz
  \shade[upper left=red,upper right=green,
         lower left=blue,lower right=yellow]
    (0,0) rectangle (3,2);
```

/tikz/lower left=⟨*color*⟩ (no default, initially `white`)

Sets the color to be used in a `bilinear interpolation` shading for the lower left corner. Also, this options selects this shading and sets the `shade` option.

/tikz/upper left=⟨*color*⟩ (no default, initially `white`)

Works like `lower left`.

/tikz/upper right=⟨*color*⟩ (no default, initially `white`)

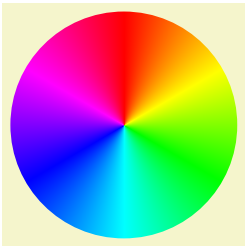
Works like `lower left`.

/tikz/lower right=⟨*color*⟩ (no default, initially `white`)

Works like `lower left`.

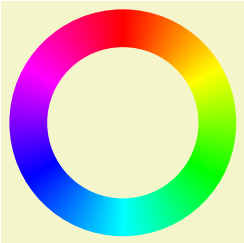
Shading color wheel

This shading fills the path with a color wheel.



```
\usepgflibrary {shadings}
\tikz \shade[shading=color wheel] (0,0) circle [radius=1.5];
```

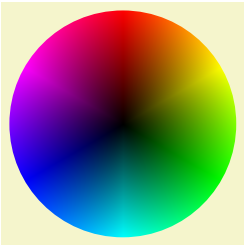
To produce a color ring, cut out a circle from the color wheel:



```
\usepgflibrary {shadings}
\tikz \shade[shading=color wheel] [even odd rule]
  (0,0) circle [radius=1.5]
  (0,0) circle [radius=1];
```

Shading `color wheel black center`

This shading looks like a color wheel, but the brightness drops to zero in the center.



```
\usepgflibrary {shadings}
\tikz \shade[shading=color wheel black center] (0,0) circle [radius=1.5];
```

Shading `color wheel white center`

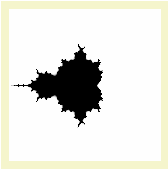
This shading looks like a color wheel, but the saturation drops to zero in the center.



```
\usepgflibrary {shadings}
\tikz \shade[shading=color wheel white center] (0,0) circle [radius=1.5];
```

Shading `Mandelbrot set`

This shading is just for fun. It fills the path with a zoomable Mandelbrot set. Note that this is *not* a bitmap graphic. Rather, the Mandelbrot set is *computed by the PDF renderer* and can be zoomed arbitrarily (give it a try, if you have a fast computer).



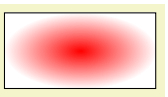
```
\usepgflibrary {shadings}
\tikz \shade[shading=Mandelbrot set] (0,0) rectangle (2,2);
```

Shading `radial`

This always-defined shading fills the path with a gradual sweep from a certain color in the middle to another color at the border. If the path is a circle, the outer color will be reached exactly at the border. If the shading is not a circle, the outer color will continue a bit towards the corners. The default inner color is gray, the default outer color is white.

`/tikz/inner color=`⟨*color*⟩ (no default)

This option sets the color used at the center of a `radial` shading. When this option is used, the `shade` and `shading=radial` options are set.



```
\usepgflibrary {shadings}
\tikz \draw[inner color=red] (0,0) rectangle (2,1);
```

/tikz/outer color=⟨*color*⟩ (no default)

This option sets the color used at the border and outside of a `radial` shading.



```
\usepgflibrary {shadings}
\tikz \draw[outer color=red,inner color=white]
  (0,0) rectangle (2,1);
```

70 Shadows Library

TikZ Library shadows

```
\usetikzlibrary{shadows} % LaTeX and plain TeX
\usetikzlibrary[shadows] % ConTeXt
```

This library defines styles that help adding a (partly) transparent shadow to a path or node.

70.1 Overview

A *shadow* is usually a black or gray area that is drawn behind a path or a node, thereby adding visual depth to a picture. The `shadows` library defines options that make it easy to add shadows to paths. Internally, these options are based on using the `preaction` option to use a path twice: Once for drawing the shadow (slightly shifted) and once for actually using the path.

Note that you can only add shadows to *paths*, not to whole scopes.

In addition to the general `shadow` option, there exist special options like `circular shadow`. These can only (sensibly) be used with a special kind of path (for `circular shadow`, a circle) and, thus, they are not as general. The advantage is, however, that they are more visually pleasing since these shadows blend smoothly with the background. Note that these special shadows use fadings, which few printers will support.

70.2 The General Shadow Option

The shadows are internally created by using a single option called `general shadow`. The different options like `drop shadow` or `copy shadow` only differ in the commands that they preset.

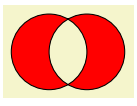
You will not need to use this option directly under normal circumstances.

`/tikz/general shadow=`⟨*shadow options*⟩ (default empty)

This option should be given to a `\path` or a `node`. It has the following effect: Before the path is used normally, it is used once with the ⟨*shadow options*⟩ in force. Furthermore, when the path is "preused" in this way, it is shifted and scaled a little bit.

In detail, the following happens: A `preaction` is used to paint the path in a special manner before it is actually painted. This "special" manner is as follows: The options in ⟨*shadow options*⟩ are used for painting this path. Typically, the ⟨*shadow options*⟩ will contain options like `fill=black` to create, say, a black shadow. Furthermore, after the ⟨*shadow options*⟩ have been set up, the following extra canvas transformations are applied to the path: It is scaled by `shadow scale` (with the origin of scaling at the path's center) and it is shifted by `shadow xshift` and `shadow yshift`.

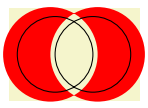
Note that since scaling and shifting is done using canvas transformations, shadows are not taken into account when the picture's bounding box is computed.



```
\usetikzlibrary {shadows}
\tikz [even odd rule]
  \draw [general shadow={fill=red}]
    (0,0) circle [radius=.5] (0.5,0) circle [radius=.5];
```

`/tikz/shadow scale=`⟨*factor*⟩ (no default, initially `1`)

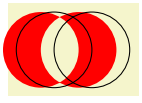
Shadows are scaled by ⟨*factor*⟩.



```
\usetikzlibrary {shadows}
\tikz [even odd rule]
  \draw [general shadow={fill=red,shadow scale=1.25}]
    (0,0) circle [radius=.5] (0.5,0) circle [radius=.5];
```

`/tikz/shadow xshift=`⟨*dimension*⟩ (no default, initially `0pt`)

Shadows are shifted horizontally by ⟨*dimension*⟩.



```
\usetikzlibrary {shadows}
\tikz [even odd rule]
  \draw [general shadow={fill=red,shadow xshift=-5pt}]
    (0,0) circle [radius=.5] (0.5,0) circle [radius=.5];
```

/tikz/shadow yshift=⟨*dimension*⟩ (no default, initially `0pt`)

Shadows are shifted vertically by ⟨*dimension*⟩.

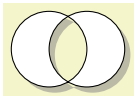
70.3 Shadows for Arbitrary Paths and Shapes

70.3.1 Drop Shadows

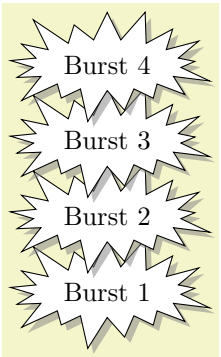
/tikz/drop shadow=⟨*shadow options*⟩ (default empty)

This option adds a drop shadow to a `\path` or a `node`. It uses the `general shadow` and passes the ⟨*shadow options*⟩ to it, plus, before them, the following extra options:

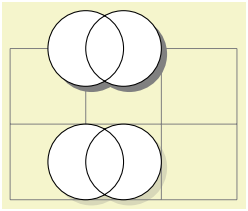
```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex,
opacity=.5, fill=black!50, every shadow
```



```
\usetikzlibrary {shadows}
\tikz [even odd rule]
  \filldraw [drop shadow,fill=white]
    (0,0) circle [radius=.5] (0.5,0) circle [radius=.5];
```




```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
  \foreach \i in {1,...,4}
    \node[starburst,drop shadow,fill=white,draw] at (0,\i) {Burst \i};
\end{tikzpicture}
```

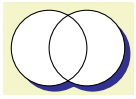


```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \filldraw [drop shadow={opacity=1},fill=white]
    (1,2)  circle [radius=.5] (1.5,2)  circle [radius=.5];

  \filldraw [drop shadow={opacity=0.25},fill=white]
    (1,.5) circle [radius=.5] (1.5,.5) circle [radius=.5];
\end{tikzpicture}
```

/tikz/every shadow (style, initially empty)

This style is executed in addition to any ⟨*shadow options*⟩ for each shadow. Use this style to reconfigure the way shadows are drawn.



```
\usetikzlibrary {shadows}
\begin{tikzpicture}[every shadow/.style={opacity=.8,fill=blue!50!black}]
  \filldraw [drop shadow,fill=white]
    (0,0) circle [radius=.5] (0.5,0) circle [radius=.5];
\end{tikzpicture}
```

70.3.2 Copy Shadows

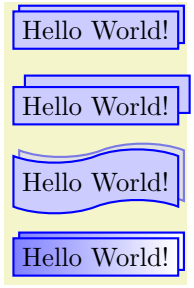
A *copy shadow* is not really a shadow. Rather, it looks like another copy of the path drawn behind the path and a little bit offset. This creates the visual impression of having multiple copies of the path/object present.

/tikz/copy shadow=⟨*shadow options*⟩ (default empty)

This shadow installs the following default options:

```
shadow scale=1, shadow xshift=.5ex, shadow yshift=-.5ex, every shadow
```

Furthermore, the options `fill=`⟨*fill color*⟩ and `draw=`⟨*draw color*⟩ are also set, where the ⟨*fill color*⟩ and ⟨*draw color*⟩ are the fill and draw colors used for the main path.




```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
  \node [copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};

  \node at (0,-1) [copy shadow={shadow xshift=1ex,shadow yshift=1ex},
                   fill=blue!20,draw=blue,thick]
    {Hello World!};

  \node at (0,-2) [copy shadow={opacity=.5},tape,
                   fill=blue!20,draw=blue,thick]
    {Hello World!};

  % We have to repeat the left color since shadings are not
  % automatically applied to shadows
  \node at (0,-3) [copy shadow={left color=blue!50},
                   left color=blue!50,draw=blue,thick]
    {Hello World!};
\end{tikzpicture}
```

`/tikz/double copy shadow=`⟨*shadow options*⟩ (default empty)

This shadow works like a `copy shadow`, only the shadow is added twice, the second time with the double `xshift` and `yshift`.




```
\usetikzlibrary {shadows,shapes.symbols}
\begin{tikzpicture}
  \node [double copy shadow,fill=blue!20,draw=blue,thick] {Hello World!};

  \node at (0,-1) [double copy shadow={shadow xshift=1ex,shadow yshift=1ex},
                   fill=blue!20,draw=blue,thick]
    {Hello World!};

  \node at (0,-2) [double copy shadow={opacity=.5},tape,
                   fill=blue!20,draw=blue,thick]
    {Hello World!};

  \node at (0,-3) [double copy shadow={left color=blue!50},
                   left color=blue!50,draw=blue,thick]
    {Hello World!};
\end{tikzpicture}
```

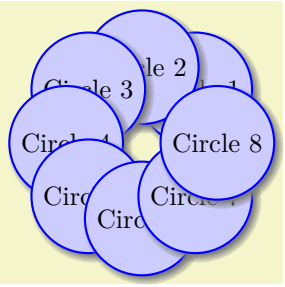
70.4 Shadows for Special Paths and Nodes

The shadows in this section should normally be added only to paths that have a special shape. They will look strange with other shapes.

`/tikz/circular drop shadow=`⟨*shadow options*⟩ (no default)

This shadow works like a drop shadow, only it adds a circular fading to the shadow. This means that the shadow will fade out at the border. The following options are preset for this shadow:

```
shadow scale=1.1, shadow xshift=.3ex, shadow yshift=-.3ex,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```

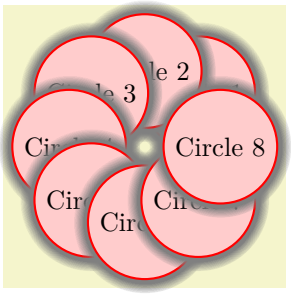



```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
    \node[circle,circular drop shadow,draw=blue,fill=blue!20,thick]
      at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

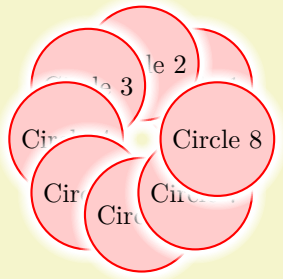
`/tikz/circular glow`=⟨*shadow options*⟩ (no default)

This shadow works much like the `circular shadow`, only it is not shifted. This creates a visual effect of a "glow" behind the circle. The following options are preset for this shadow:

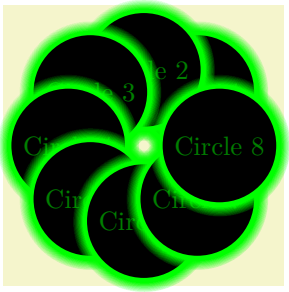
```
shadow scale=1.25, shadow xshift=0pt, shadow yshift=0pt,
fill=black, path fading={circle with fuzzy edge 15 percent},
every shadow,
```




```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow,fill=red!20,draw=red,thick]
    at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

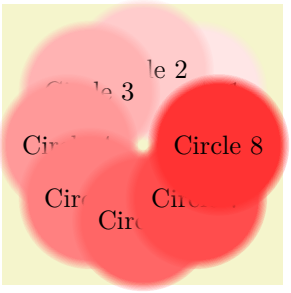



```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=white},fill=red!20,draw=red,thick]
    at (\i*45:1) {Circle \i};
\end{tikzpicture}
```




```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=green},fill=black,text=green!50!black]
    at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

An especially interesting effect can be achieved by only using the glow and not filling the path:




```
\usetikzlibrary {shadows}
\begin{tikzpicture}
  \foreach \i in {1,...,8}
  \node[circle,circular glow={fill=red!\i0}]
    at (\i*45:1) {Circle \i};
\end{tikzpicture}
```

71 Shape Library

71.1 Overview

In addition to the standard shapes `rectangle`, `circle` and `coordinate`, there exist a number of additional shapes defined in different shape libraries. Most of these shapes have been contributed by Mark Wibrow. In the present section, these shapes are described. Note that the library `shapes` is provided for compatibility only. Please include sublibraries like `shapes.geometric` or `shapes.misc` directly.

The appearance of shapes is influenced by numerous parameters like `minimum height` or `inner xsep`. These general parameters are documented in Section 17.2.3

In all of the examples presented in this section, the following `shape example` style is used:

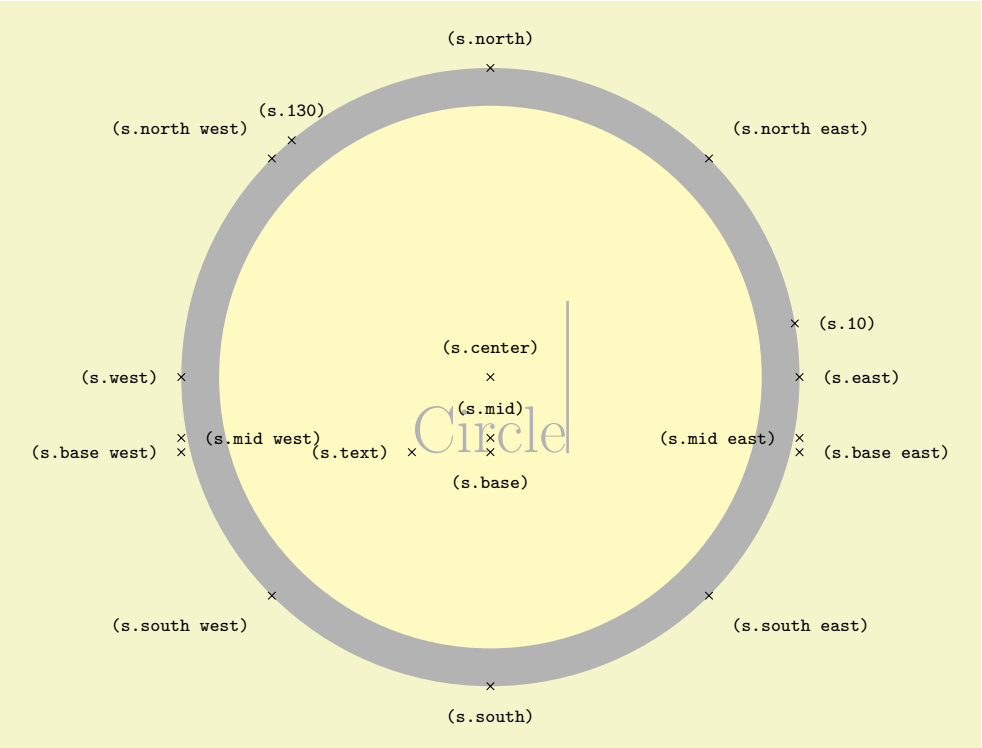
```
\tikzset{
  shape example/.style= {color      = black!30,
                         draw,
                         fill       = yellow!30,
                         line width =  .5cm,
                         inner xsep = 2.5cm,
                         inner ysep = 0.5cm}
}
```

71.2 Predefined Shapes

The three shapes `rectangle`, `circle`, and `coordinate` are always defined and no library needs to be loaded for them. While the `coordinate` shape defines only the `center` anchor, the other two shapes define a standard set of anchors.

Shape `circle`

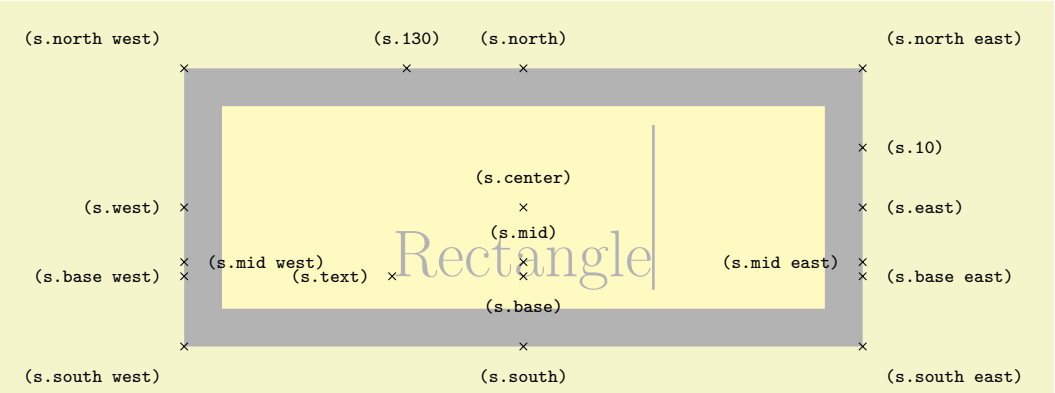
This shape draws a tightly fitting circle around the text. The following figure shows the anchors this shape defines; the anchors `10` and `130` are example of border anchors.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle,shape example] {Circle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape rectangle

This shape, which is the standard, is a rectangle around the text. The inner and outer separations (see Section 17.2.3) influence the white space around the text. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle,shape example] {Rectangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.3 Geometric Shapes

TikZ Library shapes.geometric

```
\usepgflibrary{shapes.geometric}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.geometric]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.geometric} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.geometric] % ConTeXt when using TikZ
```

This library defines different shapes that correspond to basic geometric objects like ellipses or polygons.

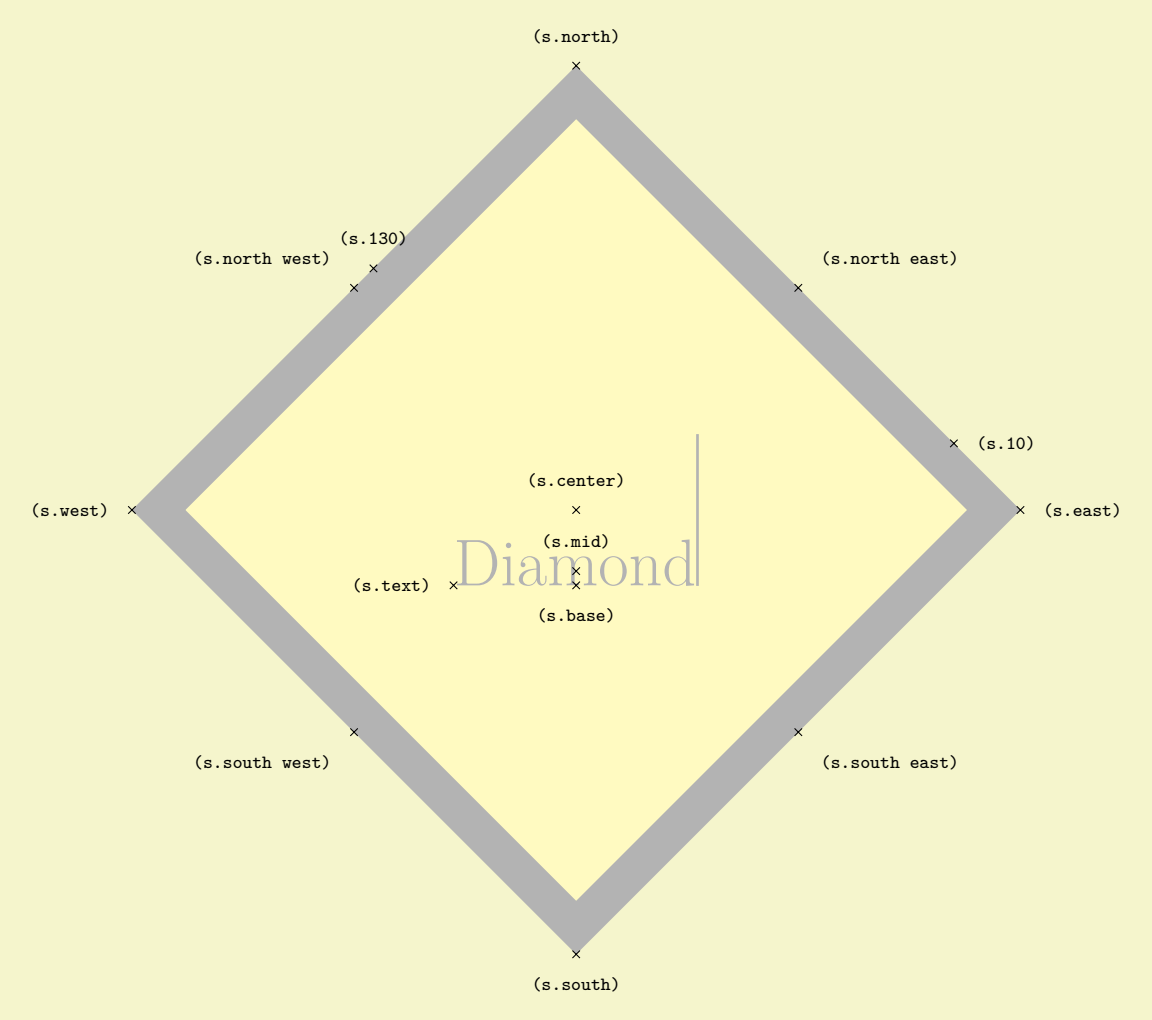
Shape diamond

This shape is a diamond tightly fitting the text box. The ratio between width and height is 1 by default, but can be changed by setting the shape aspect ratio using the following PGF key (to use this key in TikZ simply remove the /pgf/ path).

/pgf/aspect=⟨*value*⟩ (no default, initially 1.0)

The aspect is a recommendation for the quotient of the width and the height of a shape. This key calls the macro \pgfsetshapeaspect.

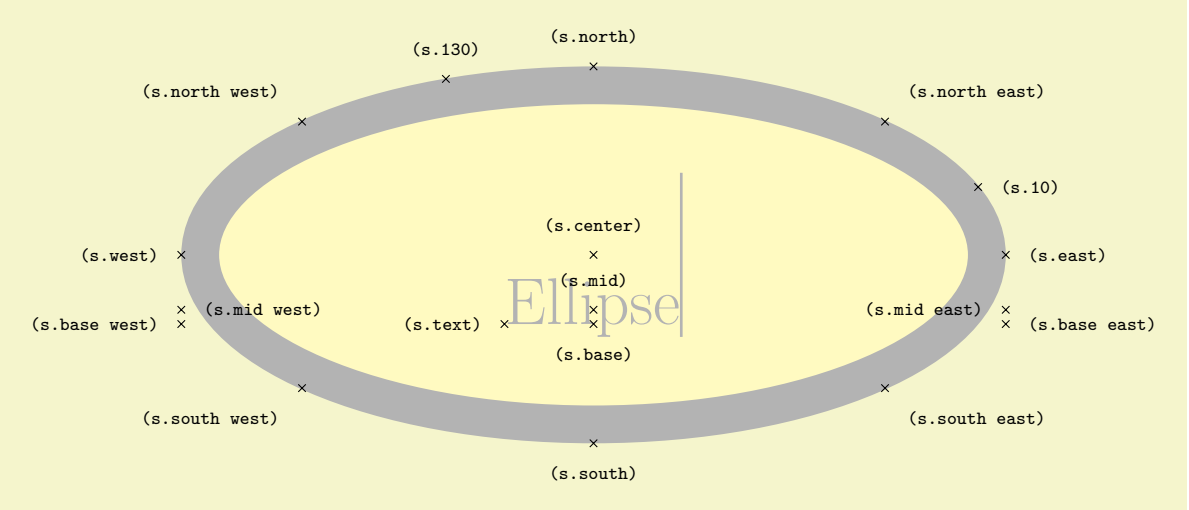
The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=diamond,shape example] {Diamond\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid/above,
     base/below,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape ellipse

This shape is an ellipse tightly fitting the text box, if no inner separation is given. The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.

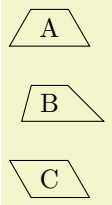



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse,shape example] {Ellipse\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `trapezium`

This shape is a trapezium, that is, a quadrilateral with a single pair of parallel lines (this can sometimes be known as a trapezoid). The trapezium shape supports the rotation of the shape border, as described in Section 17.2.3.

The lower internal angles at the lower corners of the trapezium can be specified independently, and the resulting extensions are in addition to the natural dimensions of the node contents (which includes any `inner sep`.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[every node/.style={trapezium, draw}]
   \node at (0,2) {A};
   \node[trapezium left angle=75, trapezium right angle=45]
         at (0,1) {B};
   \node[trapezium left angle=120, trapezium right angle=60]
         at (0,0) {C};
\end{tikzpicture}
```

The PGF keys to set the lower internal angles of the trapezium are shown below. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/trapezium left angle=`⟨*angle*⟩ (no default, initially `60`)

Sets the lower internal angle of the left side.

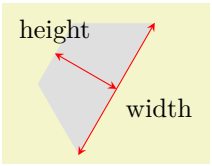
`/pgf/trapezium right angle=`⟨*angle*⟩ (no default, initially `60`)

Sets the lower internal angle of the right side.

`/pgf/trapezium angle=`⟨*angle*⟩ (style, no default)

This key stores no value itself, but sets the value of the previous two keys to ⟨*angle*⟩.

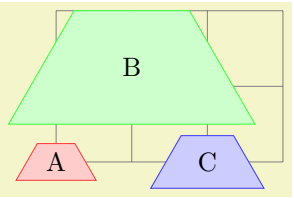
Regardless of the rotation of the shape border, the width and height of the trapezium are as follows:




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[>=stealth, every node/.style={text=black},
    shape border uses incircle, shape border rotate=60]
  \node [trapezium, fill=gray!25, minimum width=2cm] (t) {};
  \draw [red, <->] (t.bottom left corner) -- (t.bottom right corner)
    node [midway, below right] {width};
  \draw [red, <->] (t.top side) -- (t.bottom side)
    node [at start, above] {height};
\end{tikzpicture}
```

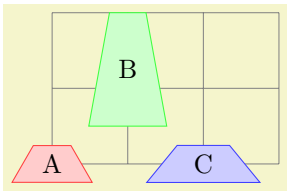
`/pgf/trapezium stretches=`⟨*boolean*⟩ (default `true`)

This key controls whether PGF allows the width and the height of the trapezium to be enlarged independently, when considering any minimum size specification. This is initially `false`, ensuring that the shape "looks the same but bigger" when enlarged.




```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \node [my node=red]                                          {A};
  \node [my node=green, minimum height=1.5cm] at (1, 1.25) {B};
  \node [my node=blue,  minimum width=1.5cm]  at (2, 0)    {C};
\end{tikzpicture}
```

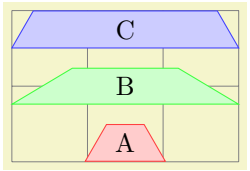
By setting ⟨*boolean*⟩ to `true`, the trapezium can be stretched horizontally or vertically.




```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
\tikzset{trapezium stretches=true}
  \draw [help lines] grid (3,2);
  \node [my node=red]                                          {A};
  \node [my node=green, minimum height=1.5cm] at (1, 1.25) {B};
  \node [my node=blue,  minimum width=1.5cm]  at (2, 0)    {C};
\end{tikzpicture}
```

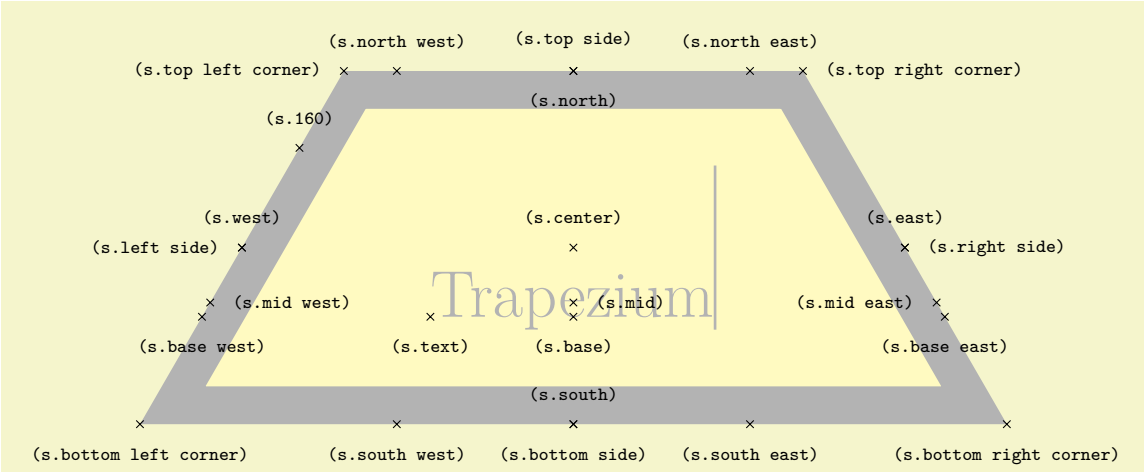
`/pgf/trapezium stretches body=`⟨*boolean*⟩ (default `true`)

This is similar to the `trapezium stretches` key except that when ⟨*boolean*⟩ is `true`, PGF enlarges only the body of the trapezium when applying minimum width.




```
\usetikzlibrary {shapes.geometric}
\tikzset{my node/.style={trapezium, fill=#1!20, draw=#1!75, text=black}}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \node [my node=red]                          at (1.5,.25)  {A};
  \node [my node=green, minimum width=3cm, trapezium stretches]
    at (1.5,1)    {B};
  \node [my node=blue,  minimum width=3cm, trapezium stretches body]
    at (1.5,1.75) {C};
\end{tikzpicture}
```

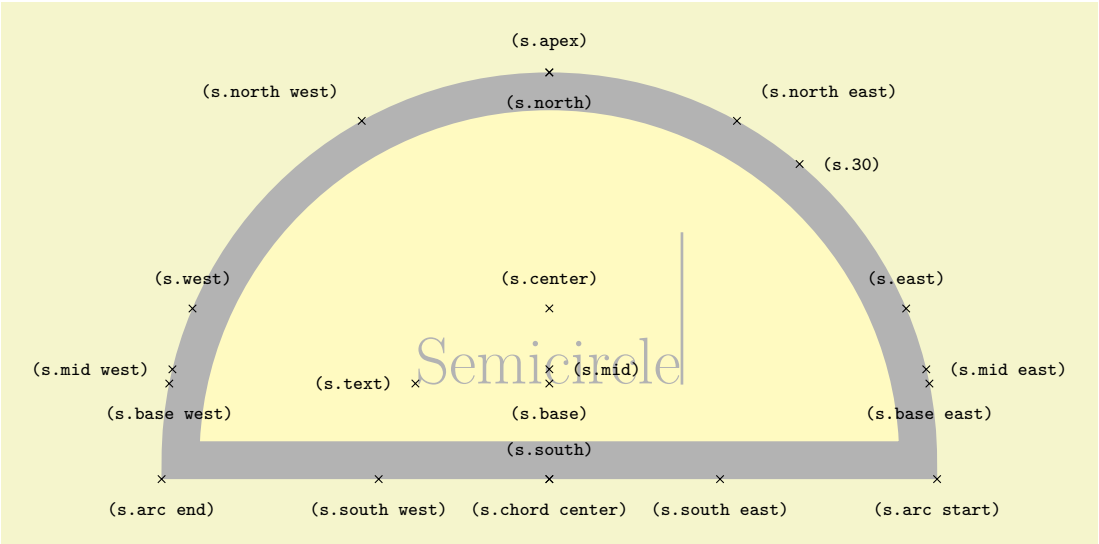
The anchors for the trapezium are shown below. The anchor `160` is an example of a border anchor.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=trapezium, shape example, inner sep=1cm]
    {Trapezium\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {bottom left corner/below, top right corner/right,
     top left corner/left,     bottom right corner/below,
     bottom side/below,        left side/left,
     right side/right,         top side/above,
     center/above,   text/below,      mid/right,       base/below,
     mid west/right, base west/below, mid east/left,   base east/below,
     west/above,     east/above,      north/below,     south/above,
     north west/above, north east/above,
     south west/below, south east/below, 160/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape semicircle

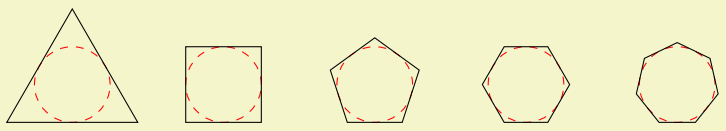
This shape is a semicircle, which tightly fits the node contents. This shape supports the rotation of the shape border, as described in Section 17.2.3. The anchors for the `semicircle` shape are shown below. Anchor `30` is an example of a border anchor.



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=semicircle,shape border rotate=0,shape example, inner sep=1cm]
    {Semicircle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {apex/above,      arc start/below, arc end/below,  chord center/below,
     center/above,    base/below,      mid/right,      text/left,
     base west/below, base east/below, mid west/left, mid east/right,
     north/below,     south/above,     east/above,     west/above,
     north west/above left, north east/above right,
     south west/below,      south east/below, 30/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

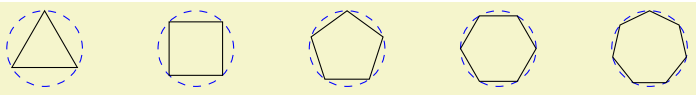
Shape regular polygon

This shape is a regular polygon, which, by default, is drawn so that a side (rather than a corner) is always at the bottom. This shape supports the rotation as described in Section 17.2.3, but the border of the polygon is *always* constructed using the incircle, whose radius is calculated to tightly fit the node contents (including any `inner sep`).



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[red, dashed] (\a*2,0)  circle[radius=0.5cm];
    \node[regular polygon, regular polygon sides=\a, draw,
     inner sep=0.3535cm] at (\a*2,0) {};
   }
\end{tikzpicture}
```

If the node is enlarged to any specified minimum size, this is interpreted as the diameter of the circumcircle, that is, the circle that passes through all the corners of the polygon border.

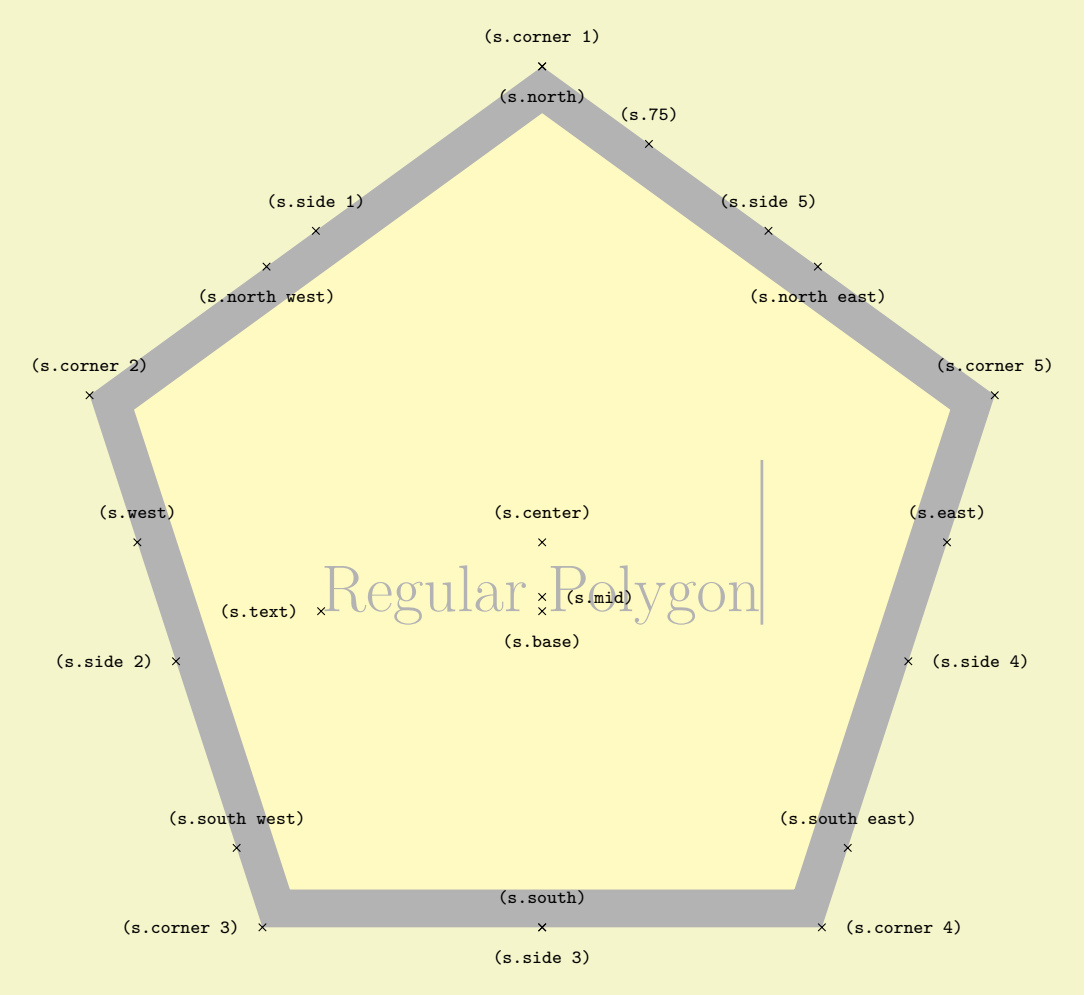


```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \foreach \a in {3,...,7}{
    \draw[blue, dashed] (\a*2,0)  circle[radius=0.5cm];
    \node[regular polygon, regular polygon sides=\a, minimum size=1cm, draw] at (\a*2,0) {};
   }
\end{tikzpicture}
```

There is a PGF key to set the number of sides for the regular polygon. To use this key in Ti*k*Z, simply remove the `/pgf/` path.

`/pgf/regular polygon sides=`⟨*integer*⟩ (no default, initially `5`)

The anchors for a regular polygon shape are shown below. The anchor `75` is an example of a border anchor.

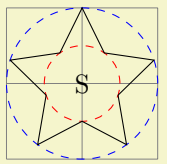



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=regular polygon, shape example, inner sep=.5cm]
    {Regular Polygon\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {corner 1/above, corner 2/above, corner 3/left, corner 4/right, corner 5/above,
     side 1/above,   side 2/left,    side 3/below,  side 4/right,   side 5/above,
     center/above, text/left,  mid/right,   base/below, 75/above,
     west/above,   east/above, north/below, south/above,
     north east/below, south east/above, north west/below, south west/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape star

This shape is a star, which by default (minus any transformations) is drawn with the first point pointing upwards. This shape supports the rotation as described in Section 17.2.3, but the border of the star is *always* constructed using the incircle.

A star should be thought of as having a set of "inner points" and "outer points". The inner points of the border are based on the radius of the circle which tightly fits the node contents, and the outer points are based on the circumcircle, the circle that passes through every outer point. Any specified minimum size, width or height, is interpreted as the diameter of the circumcircle.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
   \draw [help lines]   (0,0) grid (2,2);
   \draw [blue, dashed]  (1,1) circle[radius=1cm];
   \draw [red, dashed] (1,1) circle[radius=.5cm];
   \node [star, star point height=.5cm, minimum size=2cm, draw]
       at (1,1) {S};
\end{tikzpicture}
```

The PGF keys to set the number of star points, and the height of the star points, are shown below. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/star points`=⟨*integer*⟩ (no default, initially `5`)

Sets the number of points for the star.

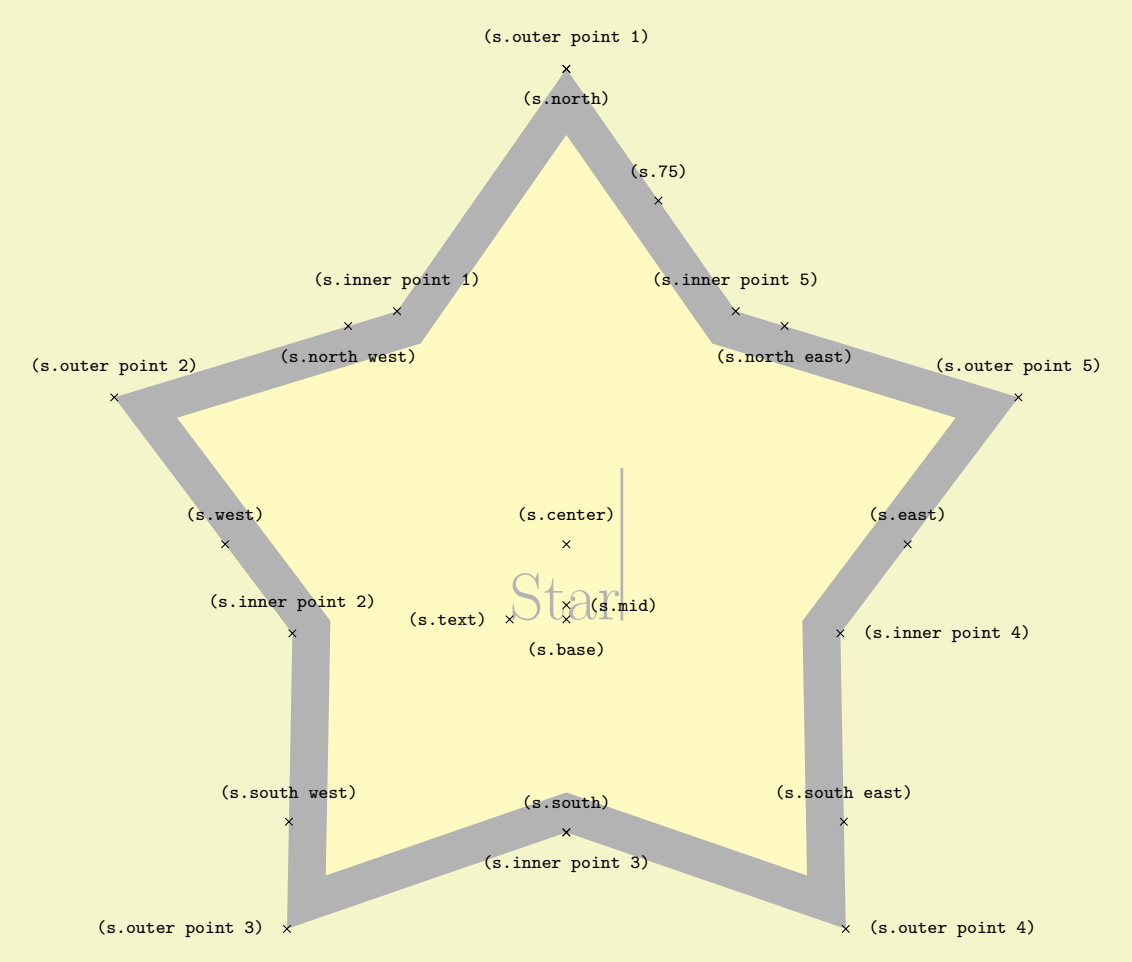
`/pgf/star point height`=⟨*distance*⟩ (no default, initially `.5cm`)

Sets the height of the star points. This is the distance between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the point height.

`/pgf/star point ratio`=⟨*number*⟩ (no default, initially `1.5`)

Sets the ratio between the inner point and outer point radii. If the star is enlarged to some specified minimum size, the inner radius is increased to maintain the ratio.

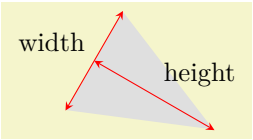
The inner and outer points form the principal anchors for the star, as shown below (anchor `75` is an example of a border anchor).



```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=star, star points=5, star point ratio=1.65, shape example, inner sep=1.5cm]
    {Star\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {inner point 1/above, inner point 2/above, inner point 3/below, inner point 4/right,
     inner point 5/above, outer point 1/above, outer point 2/above, outer point 3/left,
     outer point 4/right, outer point 5/above,
     center/above, text/left,  mid/right,   base/below, 75/above,
        west/above,   east/above, north/below, south/above,
        north east/below, south east/above, north west/below, south west/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape isosceles triangle

This shape is an isosceles triangle, which supports the rotation of the shape border, as described in Section 17.2.3. The angle of rotation determines the direction in which the apex of the triangle points (provided no other transformations are applied). However, regardless of the rotation of the shape border, the width and height are always considered as follows:




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[>=stealth, every node/.style={text=black},
    shape border uses incircle, shape border rotate=-30]
  \node [isosceles triangle, fill=gray!25, minimum width=1.5cm] (t) {};
  \draw [red, <->] (t.left corner) -- (t.right corner)
    node [midway, above left] {width};
  \draw [red, <->] (t.apex) -- (t.lower side)
    node [midway, above right] {height};
\end{tikzpicture}
```

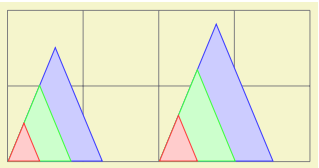
There are PGF keys to customize this shape. To use these keys in TikZ, simply remove the /pgf/ path.

/pgf/isosceles triangle apex angle=⟨*angle*⟩ (no default, initially 30)

Sets the angle of the apex of the isosceles triangle.

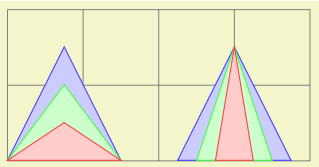
/pgf/isosceles triangle stretches=⟨*boolean*⟩ (default false)

By default ⟨*boolean*⟩ is false. This means, that when applying any minimum width or minimum height requirements, increasing the height will increase the width (and vice versa), in order to keep the apex angle the same.



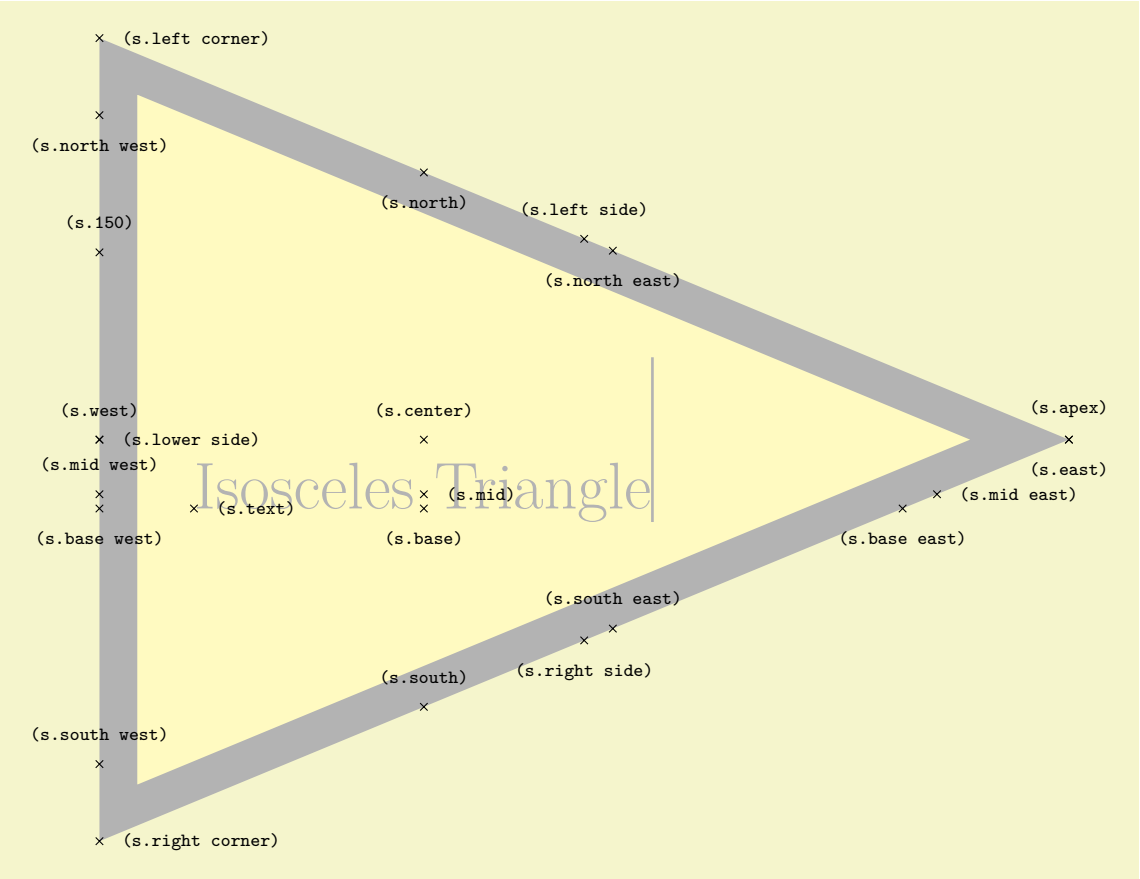
```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[paint/.style={draw=#1!75, fill=#1!20}]
  \tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
    anchor=left corner, shape border rotate=90}}
  \draw[help lines] grid(4,2);
  \foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
    \node[paint=\c, minimum height=\a cm] at (0,0) {};
    \node[paint=\c, minimum width=\a cm] at (2,0) {};
  }
\end{tikzpicture}
```

However, by setting ⟨*boolean*⟩ to true, minimum width and height can be applied independently.



```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[paint/.style={draw=#1!75, fill=#1!20}]
  \tikzset{every node/.style={isosceles triangle, draw, inner sep=0pt,
     anchor=south, shape border rotate=90, isosceles triangle stretches}}
  \draw[help lines] grid(4,2);
  \foreach \a/\c in {1.5/blue, 1/green, 0.5/red}{
    \node[paint=\c, minimum height=\a cm, minimum width=1.5cm] at (0.75,0) {};
    \node[paint=\c, minimum width=\a cm, minimum height=1.5cm] at (3,0)    {};
  }
\end{tikzpicture}
```

The anchors for the `isosceles triangle` are shown below (anchor `150` is an example of a border anchor). Note that, somewhat confusingly, the anchor names such as `left side` and `right corner` are named as if the triangle is rotated to 90 degrees. Note also that the `center` anchor does not necessarily correspond to any kind of geometric center.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=isosceles triangle, shape example, inner xsep=1cm]
    {Isosceles Triangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {apex/above,      left corner/right, right corner/right,
     left side/above, right side/below,  lower side/right,
     center/above,    text/right,        150/above,
     mid/right,       mid west/above,    mid east/right,
     base/below,      base west/below,   base east/below,
     west/above, east/below, north/below, south/above,
     north west/below, north east/below,
     south west/above, south east/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `kite`

This shape is a kite, which supports the rotation of the shape border, as described in Section 17.2.3. There are PGF keys to specify the upper and lower vertex angles of the kite. To use these keys in TikZ, simply remove the `/pgf/` path.

`/pgf/kite upper vertex angle=`⟨*angle*⟩ (no default, initially `120`)

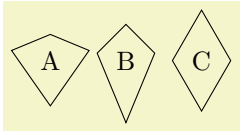
Sets the upper internal angle of the kite.

`/pgf/kite lower vertex angle=`⟨*angle*⟩ (no default, initially `60`)

Sets the lower internal angle of the kite.

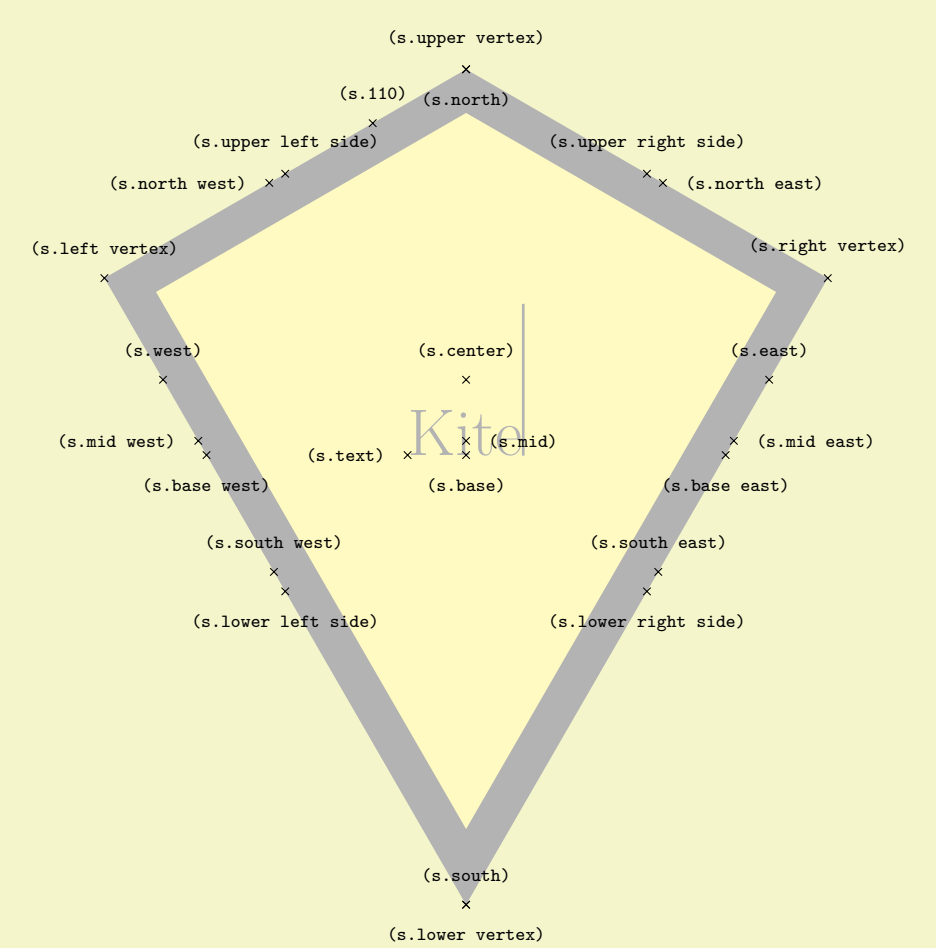
`/pgf/kite vertex angles`=⟨*angle specification*⟩ (no default)

This key sets the keys for both the upper and lower vertex angles (it stores no value itself). ⟨*angle specification*⟩ can be pair of angles in the form ⟨*upper angle*⟩ `and` ⟨*lower angle*⟩, or a single angle. In this latter case, both the upper and lower vertex angles will be the same.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[every node/.style={kite, draw}]
  \node[kite upper vertex angle=135, kite lower vertex angle=70] at (0,0) {A};
  \node[kite vertex angles=90 and 45] at (1,0) {B};
  \node[kite vertex angles=60]        at (2,0) {C};
\end{tikzpicture}
```

The anchors for the `kite` are shown below. Anchor `110` is an example of a border anchor.

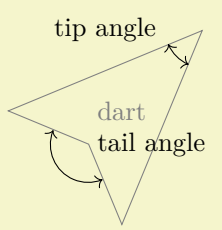


```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=kite, shape example, inner sep=1.5cm]
    {Kite\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {upper vertex/above, left vertex/above,    lower vertex/below,
     right vertex/above, upper left side/above, upper right side/above,
     lower left side/below, lower right side/below,
     center/above,   text/left,       mid/right,        base/below,
     mid west/left,  base west/below, mid east/right,   base east/below,
     west/above,     east/above,      north/below,     south/above,
     north west/left, north east/right,
     south west/above, south east/above, 110/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape dart

This shape is a dart (which can also be known as an arrowhead or concave kite). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of the border rotation determines the direction in which the dart points (unless other transformations have been applied).

There are PGF keys to set the angle for the 'tip' of the dart and the angle between the 'tails' of the dart. To use these keys in Ti*k*Z, simply remove the /pgf/ path.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
   \node[dart, draw, gray, shape border uses incircle, shape border rotate=45]
       (d) {dart};
   \draw [<->] (d.tip)++(202.5:.5cm) arc(202.5:247.5:.5cm);
   \node [left=.5cm] at (d.tip) {tip angle};
   \draw [<->] (d.tail center)++(157.5:.5cm) arc(157.5:292.5:.5cm);
   \node [right] at (d.tail center) {tail angle};
\end{tikzpicture}
```

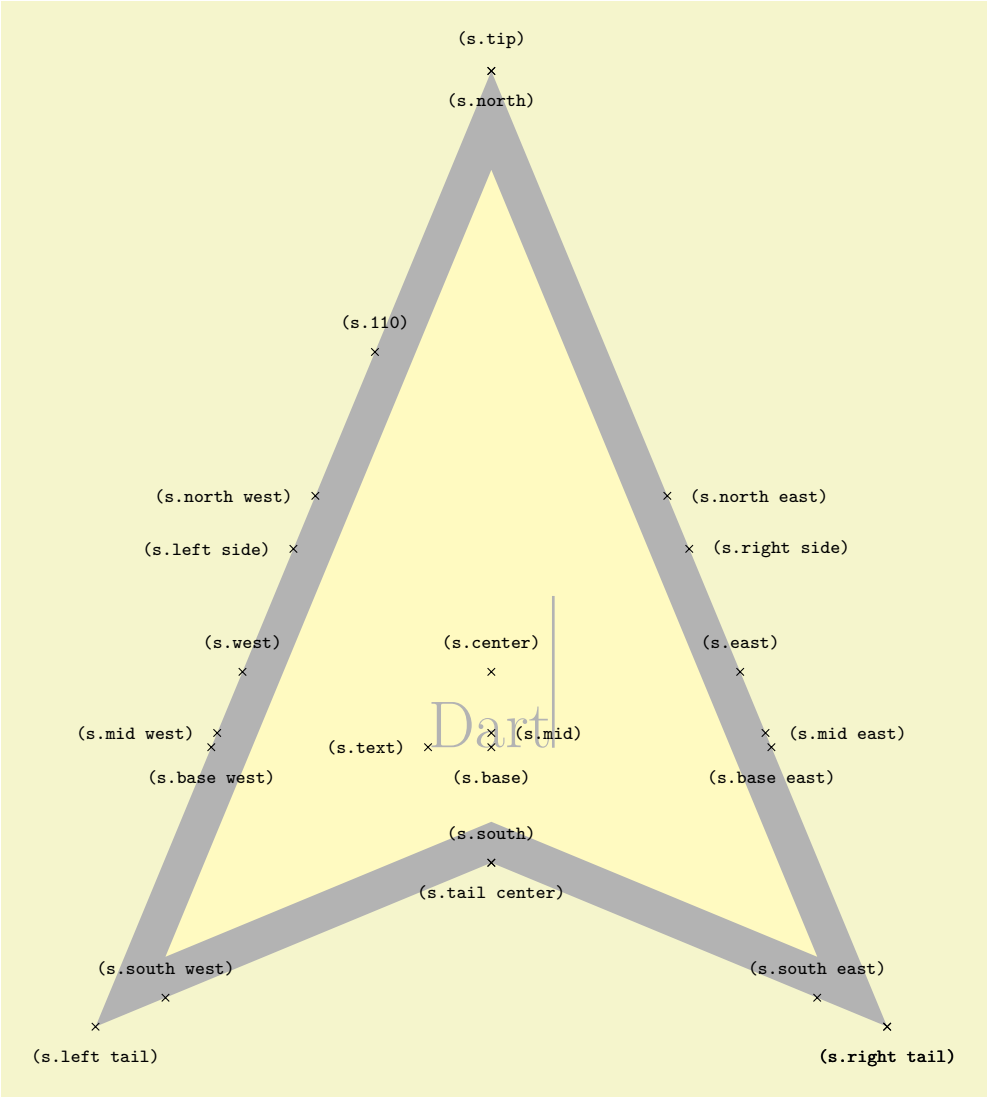
/pgf/dart tip angle=⟨*angle*⟩ (no default, initially 45)

Sets the angle at the tip of the dart.

/pgf/dart tail angle=⟨*angle*⟩ (no default, initially 135)

Sets the angle between the tails of the dart.

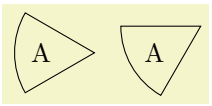
The anchors for the dart shape are shown below (note that the shape is rotated 90 degrees anti-clockwise). Anchor 110 is an example of a border anchor.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=dart, shape border rotate=90, shape example, inner sep=1.25cm]
    {Dart\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {tip/above,       tail center/below, right tail/below,
     left tail/below, right tail/below,  left side/left,   right side/right,
     center/above,    text/left,         mid/right,        base/below,
     mid west/left,   base west/below,   mid east/right,   base east/below,
     west/above,      east/above,        north/below,      south/above,
     north west/left, north east/right,  south west/above, south east/above,
     110/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `circular sector`

This shape is a circular sector (which can also be known as a wedge). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of the border rotation determines the direction in which the 'apex' of the sector points (unless other transformations have been applied).



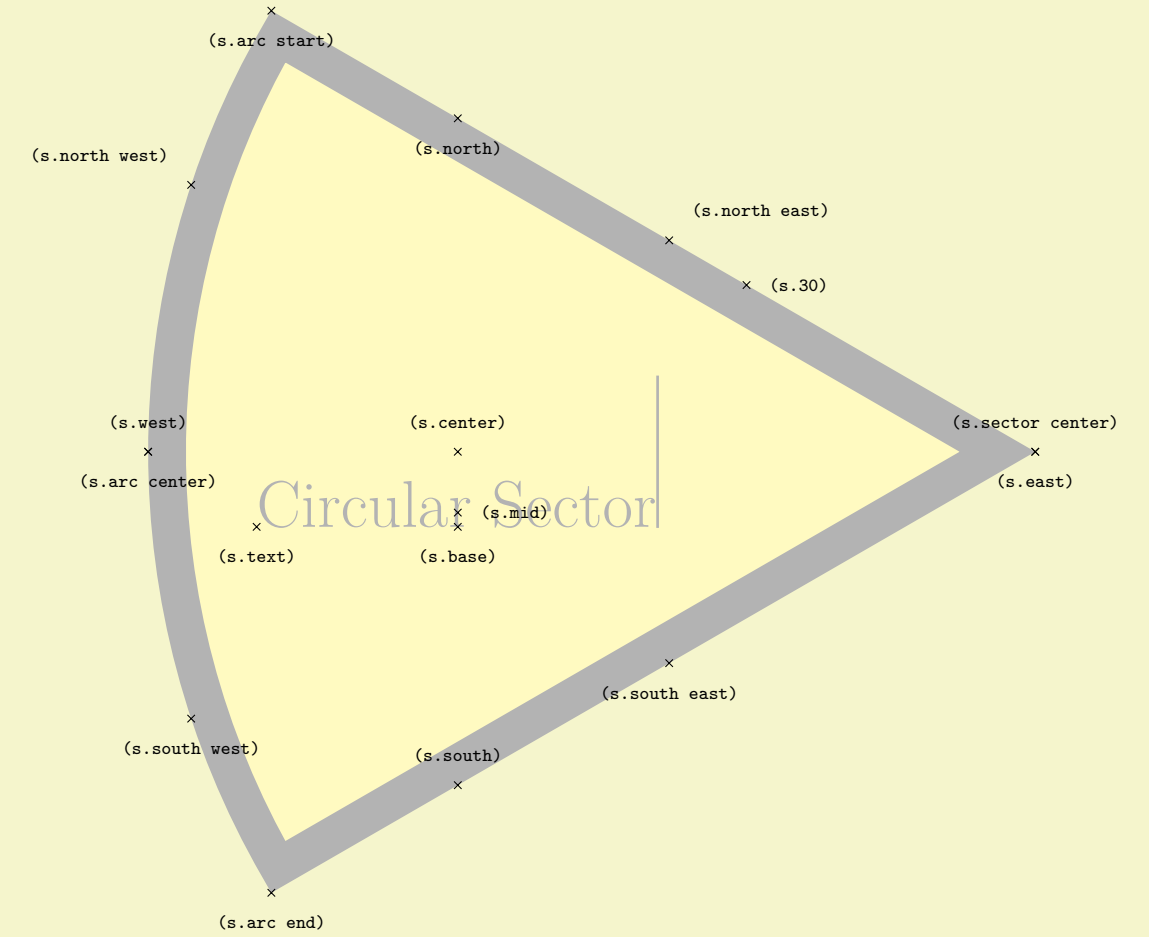

```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[
   every node/.style={circular sector, shape border uses incircle, draw},
]
   \node at (0,0) {A};
   \node [shape border rotate=30] at (1.5,0) {A};
\end{tikzpicture}
```

There is a PGF key to set the central angle of the sector, which is expected to be less than 180 degrees. To use this key in Ti*k*Z, simply remove the `/pgf/` path.

`/pgf/circular sector angle=`⟨*angle*⟩ (no default, initially `60`)

Sets the central angle of the sector.

The anchors for the circular sector shape are shown below. Anchor `30` is an example of a border anchor.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circular sector,  style=shape example, inner sep=1cm]
      {Circular Sector\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
   {sector center/above, arc start/below, arc end/below, arc center/below,
    center/above,        base/below,      mid/right,     text/below,
    north/below,         south/above,     east/below,    west/above,
    north west/above left, north east/above right,
    south west/below,      south east/below, 30/right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

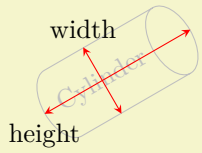
Shape cylinder

This shape is a 2-dimensional representation of a cylinder, which supports the rotation of the shape border as described in Section 17.2.3.



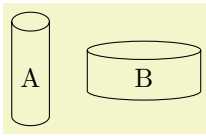

```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}
  \node[cylinder, draw, shape aspect=.5] {ABC};
\end{tikzpicture}
```

Regardless the rotation of the shape border, the height is always the distance between the curved ends, and the width is always the distance between the straight sides.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[>=stealth]
  \node [cylinder, gray!50, rotate=30, draw,
    minimum height=2cm, minimum width=1cm] (c) {Cylinder};
  \draw[red, <->] (c.top)    -- (c.bottom)
    node [at end, below, black]   {height};
  \draw[red, <->] (c.north) -- (c.south)
    node [at start, above, black] {width};
\end{tikzpicture}
```

Enlarging the shape to some minimum height will stretch only the body of the cylinder. By contrast, enlarging the shape to some minimum width will stretch the curved ends.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[shape aspect=.5]
  \tikzset{every node/.style={cylinder, shape border rotate=90, draw}}
  \node [minimum height=1.5cm]            {A};
  \node [minimum width=1.5cm]  at (1.5,0) {B};
\end{tikzpicture}
```

There are various keys to customize this shape (to use PGF keys in TikZ, simply remove the /pgf/ path).

/pgf/aspect=⟨*value*⟩ (no default, initially 1.0)

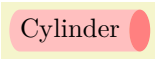
The aspect is a recommendation for the quotient of the radii of the cylinder end. This may be ignored if the shape is enlarged to some minimum width.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[]
  \tikzset{every node/.style={cylinder, shape border rotate=90, draw}}
  \node [aspect=1.0]           {A};
  \node [aspect=0.5]  at (1,0) {B};
  \node [aspect=0.25] at (2,0) {C};
\end{tikzpicture}
```

/pgf/cylinder uses custom fill=⟨*boolean*⟩ (default true)

This enables the use of a custom fill for the body and the end of the cylinder. The background path for the shape should not be filled (e.g., in TikZ, the fill option for the node must be implicitly or explicitly set to none). Internally, this key sets the TEX-if \ifpgfcylinderusescustomfill appropriately.




```
\usetikzlibrary {shapes.geometric}
\begin{tikzpicture}[aspect=0.5]
  \node [cylinder, cylinder uses custom fill, cylinder end fill=red!50,
         cylinder body fill=red!25] {Cylinder};
\end{tikzpicture}
```

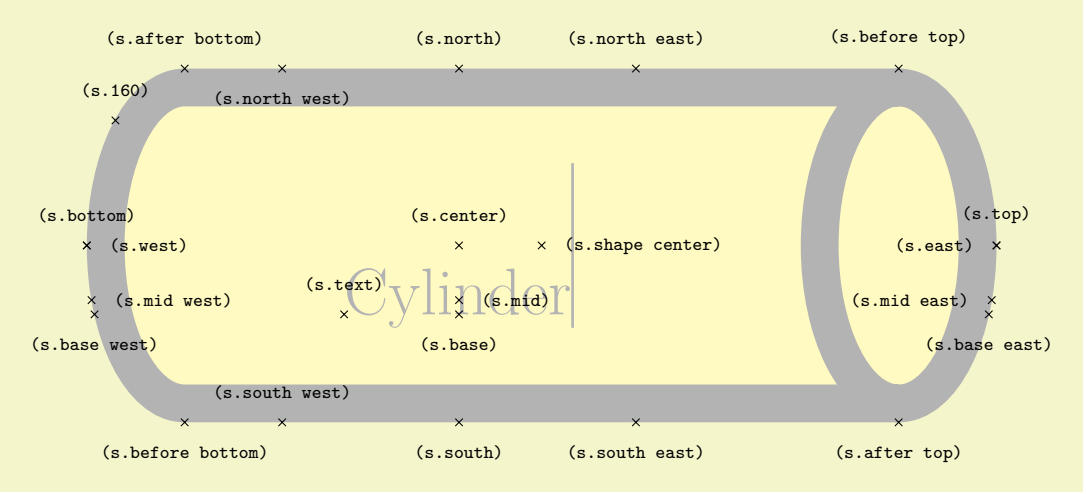
/pgf/cylinder end fill=⟨*color*⟩ (no default, initially white)

Sets the color for the end of the cylinder.

/pgf/cylinder body fill=⟨*color*⟩ (no default, initially white)

Sets the color for the body of the cylinder.

The anchors of this shape are shown below (anchor `160` is an example of a border anchor). Note that the cylinder shape does not distinguish between `outer xsep` and `outer ysep`. Only the larger of the two values is used for the shape. Note also the difference between the `center` and `shape center` anchors: `center` is the center of the cylinder body and also the center of rotation. The `shape center` is the center of the shape which includes the 2-dimensional representation of the cylinder top.




```
\usetikzlibrary {shapes.geometric}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=cylinder, shape example, aspect=.5, inner xsep=3cm,
        inner ysep=1cm] {Cylinder\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {before top/above,    top/above,       after top/below,
     before bottom/below, bottom/above,    after bottom/above,
     mid/right,           mid west/right,  mid east/left,
     base/below,          base west/below, base east/below,
     center/above,        text/above,      shape center/right,
     west/right, east/left, north/above, south/below,
     north west/below, north east/above,
     south west/above, south east/below, 160/above}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.4 Symbol Shapes

TikZ Library shapes.symbols

```
\usepgflibrary{shapes.symbols}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.symbols]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.symbols} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.symbols] % ConTeXt when using TikZ
```

This library defines shapes that can be used for drawing symbols like a forbidden sign or a cloud.

Shape correct forbidden sign

This shape places the node inside a circle with a diagonal from the upper left to the lower right added. The circle is part of the background, the diagonal line part of the foreground path; thus, the diagonal line is on top of the text.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [correct forbidden sign,line width=1ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape forbidden sign

This shape is like `correct forbidden sign`, only the line goes from the lower left to the upper right. The strange naming of these shapes is for historical reasons.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [forbidden sign,line width=1ex,draw=red,fill=white] {Smoking};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape magnifying glass

This shape places the node inside a circle with a handle attached to the node. The angle of the handle and its length can be adjusted using two keys:

`/pgf/magnifying glass handle angle=`⟨*degree*⟩ (default `-45`)

The angle of the handle.

`/pgf/magnifying glass handle aspect=`⟨*factor*⟩ (default `1.5`)

The length of the handle as a multiple of the circle radius.

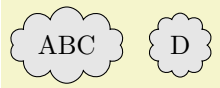



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node [magnifying glass,line width=1ex,draw] {huge};
\end{tikzpicture}
```

The shape inherits all anchors from the `circle` shape.

Shape cloud

This shape is a cloud, drawn to tightly fit the node contents (strictly speaking, using an ellipse which tightly fits the node contents – including any `inner sep`).




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node[cloud, draw, fill=gray!20, aspect=2] {ABC};
  \node[cloud, draw, fill=gray!20] at (1.5,0) {D};
\end{tikzpicture}
```

A cloud should be thought of as having a number of "puffs", which are the individual arcs drawn around the border. There are PGF keys to specify how the cloud is drawn (to use these keys in Ti*k*Z, simply remove the `/pgf/` path).

`/pgf/cloud puffs=`⟨*integer*⟩ (no default, initially `10`)

Sets the number of puffs for the cloud.

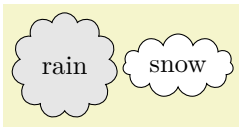
`/pgf/cloud puff arc=`⟨*angle*⟩ (no default, initially `135`)

Sets the length of the puff arc (in degrees). A shorter arc can produce better looking joins between puffs for larger line widths.

Like the diamond shape, the cloud shape also uses the `aspect` key to determine the ratio of the width and the height of the cloud. However, there may be circumstances where it may be undesirable to continually specify the `aspect` for the cloud. Therefore, the following key is implemented:

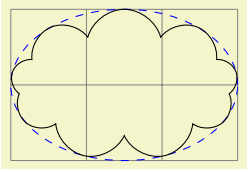
`/pgf/cloud ignores aspect=`⟨*boolean*⟩ (default `true`)

Instruct PGF to ignore the `aspect` key. Internally, the TEX-if `\ifpgfcloudignoresaspect` is set appropriately. The initial value is `false`.



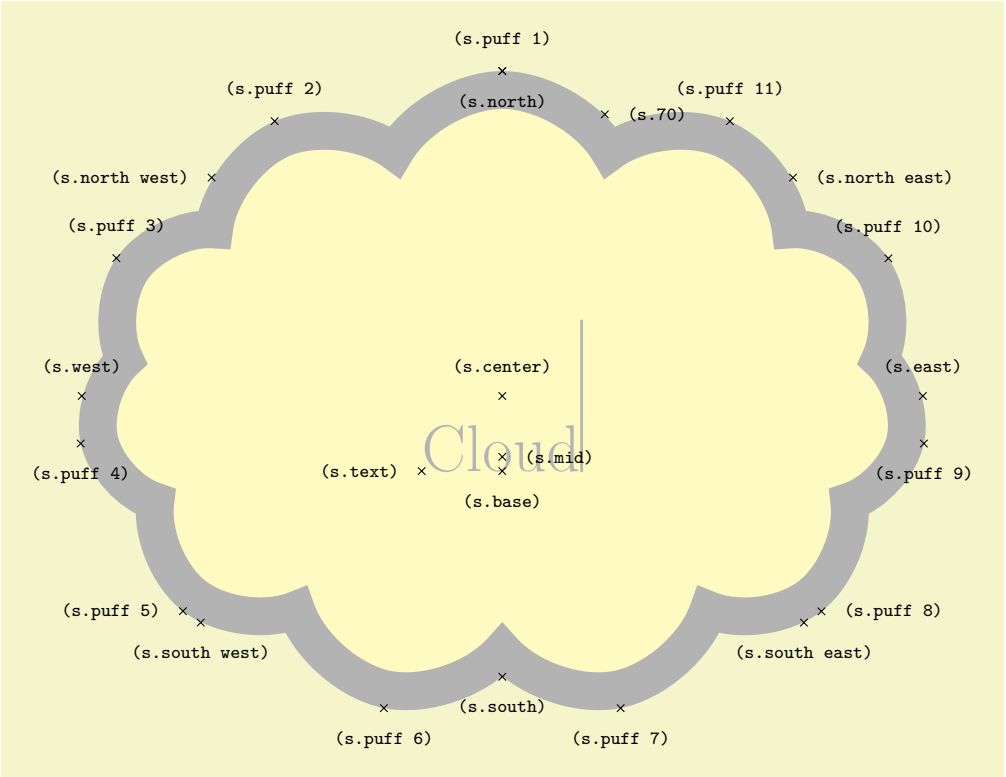

```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[aspect=1, every node/.style={cloud, cloud puffs=11, draw}]
  \node [fill=gray!20]                                  {rain};
  \node [cloud ignores aspect, fill=white] at (1.5,0) {snow};
\end{tikzpicture}
```

Any minimum size requirements are applied to the "circum-ellipse", which is the ellipse which passes through all the midpoints of the puff arcs. These requirements are considered *after* any aspect specification is applied.



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \draw [blue, dashed] (1.5, 1) ellipse [x radius=1.5cm, y radius=1cm];
  \node [cloud, cloud puffs=9, draw, minimum width=3cm, minimum height=2cm]
    at (1.5, 1) {};
\end{tikzpicture}
```

The anchors for the cloud shape are shown below for a cloud with eleven puffs. Anchor 70 is an example of a border anchor.




```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=cloud, style=shape example, cloud puffs=11, aspect=1.5,
       cloud puff arc=120,inner ysep=1cm] {Cloud\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
   {puff 1/above, puff 2/above,  puff 3/above,  puff 4/below,
    puff 5/left,  puff 6/below,  puff 7/below,  puff 8/right,
    puff 9/below, puff 10/above, puff 11/above, 70/right,
    center/above, base/below,    mid/right,     text/left,
    north/below,  south/below,   east/above,    west/above,
    north west/left,             north east/right,
    south west/below,            south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape starburst

This shape is a randomly generated elliptical star, which supports the rotation of the shape border as described in Section 17.2.3.

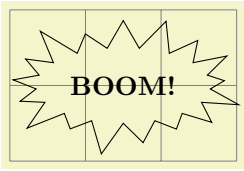



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node[starburst, fill=yellow, draw=red, line width=2pt] {\bf BANG!};
\end{tikzpicture}
```

Like the `star` shape, the starburst should be thought of as having a set of inner points and outer points. The inner points lie on the ellipse which tightly fits the node contents (including any `inner sep`).

Using a specified 'starburst point height' value, the outer points are generated randomly between this value and one quarter of this value. For a given starburst shape, the angle between each point is fixed, and is determined by the number of points specified for the starburst.

It is important to note that, whilst the maximum possible point height is used to calculate minimum width or height requirements, the outer points are randomly generated, so there is (unfortunately) no guarantee that any such requirements will be fully met.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \draw[help lines] grid(3,2);
  \node[starburst, draw, minimum width=3cm, minimum height=2cm]
    at (1.5, 1) {\bf BOOM!};
\end{tikzpicture}
```

There are PGF keys to control the drawing of the starburst shape. To use these keys in Ti*k*Z, simply remove the /pgf/ path.

/pgf/starburst points=⟨*integer*⟩ (no default, initially 17)

Sets the number of outer points for the starburst.

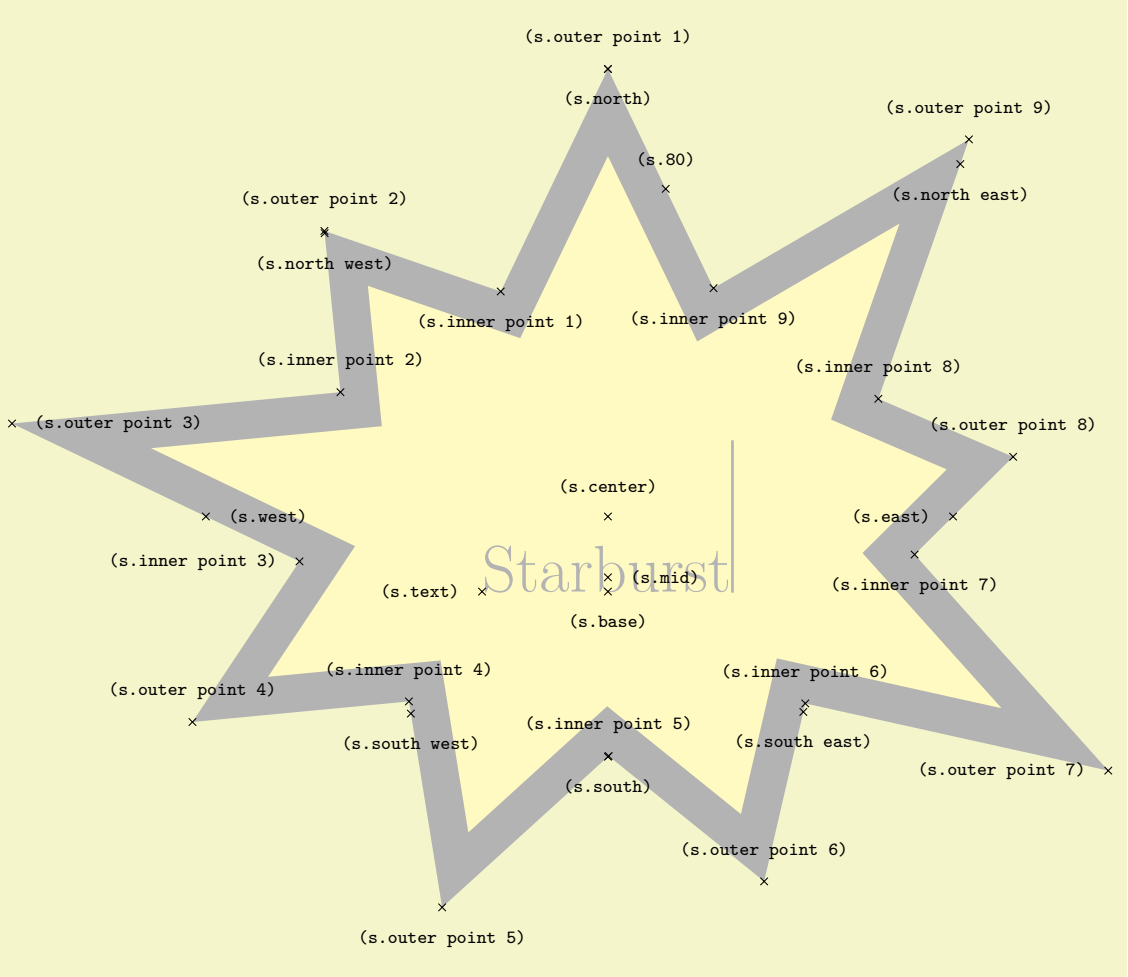
/pgf/starburst point height=⟨*length*⟩ (no default, initially .5cm)

Sets the *maximum* distance between the inner point radius and the outer point radius.

/pgf/random starburst=⟨*integer*⟩ (no default, initially 100)

Sets the seed for the random number generator for creating the starburst. The maximum value for ⟨*integer*⟩ is 16383. If ⟨*integer*⟩=0, the random number generator will not be used, and the maximum point height will be used for all outer points. If ⟨*integer*⟩ is omitted, a seed will be randomly chosen.

The basic anchors for a nine point `starburst` shape are shown below. Anchor 80 is an example of a border anchor.




```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=starburst, starburst points=9, starburst point height=3.5cm,
        style=shape example,inner sep=1cm]
    {Starburst\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {outer point 1/above, outer point 2/above, outer point 3/right,
     outer point 4/above, outer point 5/below, outer point 6/above,
     outer point 7/left,  outer point 8/above, outer point 9/above,
     inner point 1/below, inner point 2/above, inner point 3/left,
     inner point 4/above, inner point 5/above, inner point 6/above,
     inner point 7/below, inner point 8/above, inner point 9/below,
     center/above, text/left,   mid/right, base/below, 80/above,
     north/below,  south/below, east/left, west/right,
     north east/below, south west/below, south east/below, north west/below}
  \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape signal

This shape is a “signal” or sign shape, that is, a rectangle, with optionally pointed sides. A signal can point “to” somewhere, with outward points in that direction. It can also be “from” somewhere, with inward points from that direction. The resulting points extend the node contents (which include the `inner sep`).




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  [every node/.style={signal, draw,  text=white, signal to=nowhere}]
  \node[fill=green!65!black, signal to=east] at (0,1) {To East};
  \node[fill=red!65!black, signal from=east] at (0,0) {From East};
\end{tikzpicture}
```

There are PGF keys for drawing the signal shape (to use these keys in Ti*k*Z, simply remove the `/pgf/` path):

`/pgf/signal pointer angle=`⟨*angle*⟩ (no default, initially `90`)

Sets the angle for the pointed sides of the shape. This angle is maintained when enforcing any minimum size requirements, so any adjustment to the width will affect the height, and vice versa.

`/pgf/signal from=`⟨*direction*⟩ and ⟨*opposite direction*⟩ (no default, initially `nowhere`)

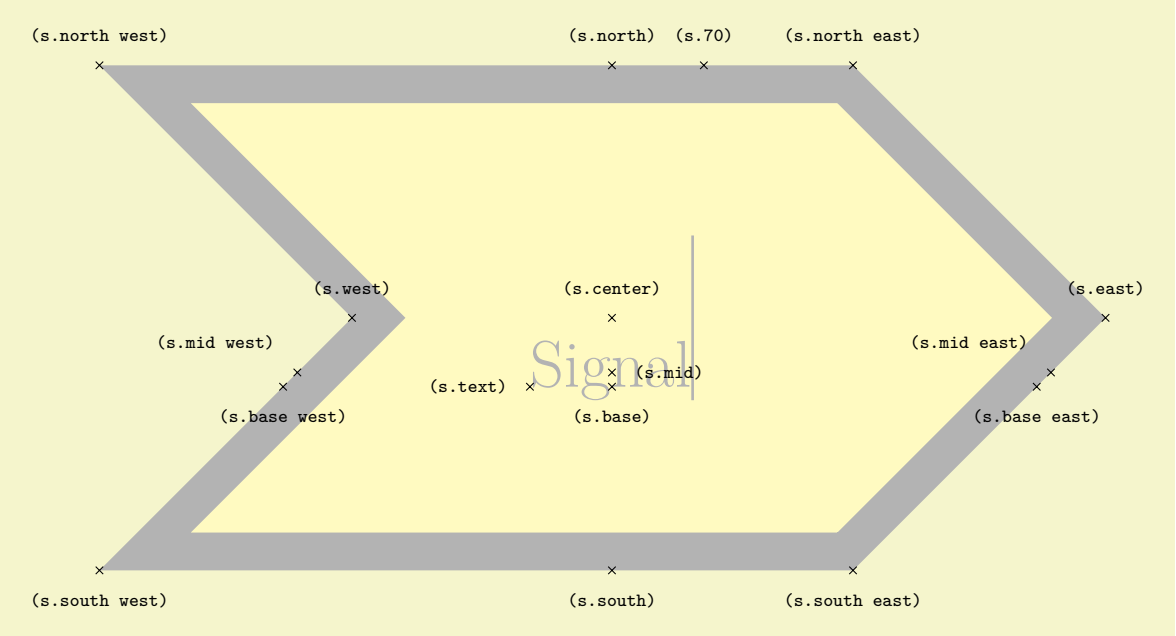
Sets which sides take an inward pointer (i.e., that points towards the center of the shape). The possible values for ⟨*direction*⟩ and ⟨*opposite direction*⟩ are the compass point directions `north`, `south`, `east` and `west` (or `above`, `below`, `right` and `left`). An additional keyword `nowhere` can be used to reset the sides so they have no pointers. When used with `signal from` key, this only resets inward pointers; used with the `signal to` key, it only resets outward pointers.

`/pgf/signal to=`⟨*direction*⟩ and ⟨*opposite direction*⟩ (no default, initially `east`)

Sets which sides take an outward pointer (i.e., that points away from the shape).

Note that PGF will ignore any instruction to use directions that are not opposites (so using the value `east and north`, will result in only `north` being assigned a pointer). This is also the case if non-opposite values are used in the `signal to` and `signal from` keys at the same time. So, for example, it is not possible for a signal to have an outward point to the left, and also have an inward point from below.

The anchors for the signal shape are shown below. Anchor `70` is an example of a border anchor.



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=signal, signal from=west, shape example, inner sep=2cm]
    {Signal\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {text/left,   center/above,    70/above,
     base/below,  base east/below, base west/below,
     mid/right,   mid east/above left,  mid west/above left,
     north/above,      south/below,
     east/above,       west/above,
     north west/above, north east/above,
     south west/below, south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape tape

This shape is a rectangle with optional, "bendy" top and bottom sides, which tightly fits the node contents (including the `inner sep`).

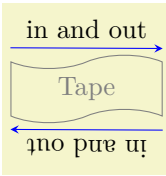



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \node[tape, draw]{ABCD};
  \node[tape, draw, tape bend top=none] at (1.5, 0) {EFGH};
\end{tikzpicture}
```

There are PGF keys to specify which sides bend and how high the bends are (to use these keys in Ti*k*Z, simply remove the `/pgf/` path):

`/pgf/tape bend top`=⟨*bend style*⟩ (no default, initially `in and out`)

Specifies how the top side bends. The ⟨*bend style*⟩ is either `in and out`, `out and in` or `none` (i.e., a straight line). The bending sides are drawn in a clockwise direction, and using the bend style `in and out` will mean the side will first bend inwards and then bend outwards. The opposite holds true for `out and in`.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[-stealth]
  \node[tape, draw, gray, minimum width=2cm](t){Tape};
  \draw [blue]([yshift=5pt] t.north west) -- ([yshift=5pt]t.north east)
        node[midway, above, black]{in and out};
  \draw [blue]([yshift=-5pt]t.south east) -- ([yshift=-5pt]t.south west)
        node[sloped, allow upside down, midway, above, black]{in and out};
\end{tikzpicture}
```

This might take a bit of getting used to, but just remember that when you want the bendy sides to be parallel, the sides take the same bend style. It is possible for the top and bottom sides to take opposite bend styles, but the author of this shape cannot think of a single use for such a combination.



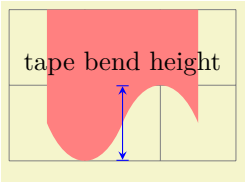

```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[every node/.style={tape, draw}]
  \node [tape bend top=out and in, tape bend bottom=out and in] {Parallel};
  \node at (2,0) [tape bend bottom=out and in]                  {Why?};
\end{tikzpicture}
```

`/pgf/tape bend bottom`=⟨*bend style*⟩ (no default, initially `in and out`)

Specifies how the bottom side bends.

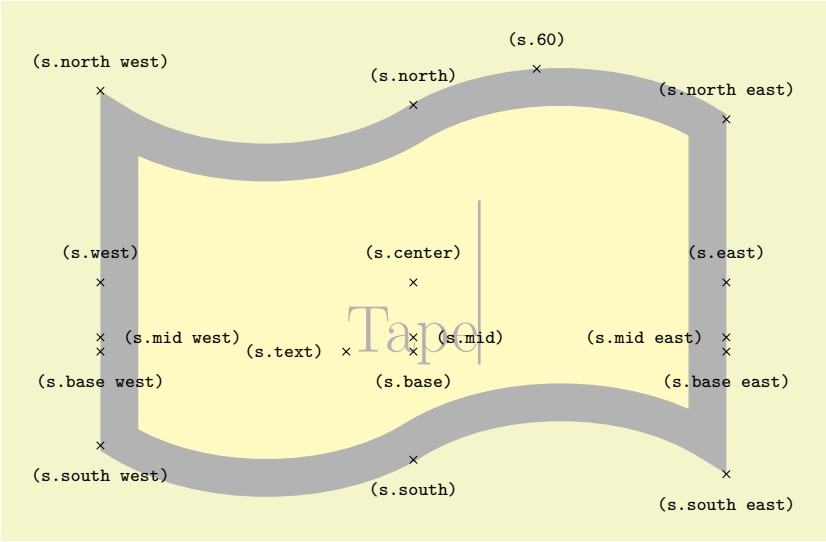
`/pgf/tape bend height`=⟨*length*⟩ (no default, initially `5pt`)

Sets the total height for a side with a bend.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[>=stealth]
  \draw [help lines] grid(3,2);
  \node [tape, fill, minimum size=2cm, red!50, tape bend top=none,
         tape bend height=1cm] at (1.5,1.5) (t) {};
  \draw [|<->|, blue] (1.5,0) -- (1.5,1)
         node [at end, above, black]{tape bend height};
\end{tikzpicture}
```

The anchors for the tape shape are shown below. Anchor `60` is an example of a border anchor. Note that border anchors will snap to the center of convex curves (i.e. when bending in).




```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=tape, tape bend height=1cm, shape example, inner xsep=3cm]
    {Tape\vrule width1pt height2cm};
   \foreach \anchor/\placement in
    {text/left,  center/above,    60/above,
     base/below, base east/below, base west/below,
     mid/right,  mid east/left,   mid west/right,
     north/above, south/below,  east/above, west/above,
     north west/above, north east/above,
     south west/below, south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `magnetic tape`

This shape represents a 'magnetic tape' or any sequential data store that is sometimes used in flowcharts. It is essentially a circle with a little tail:

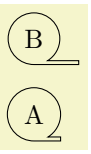



```
\usetikzlibrary {shapes.symbols}
\tikz\node [magnetic tape, draw] (A) {A};
```

The following keys can be used to customise the `magnetic tape` shape:

`/pgf/magnetic tape tail extend=`⟨*distance*⟩ (no default, initially `0cm`)

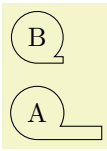
This key sets how far the tail extends beyond the radius of the tape. Negative values will be ignored.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[every node/.style={magnetic tape, draw}]
  \node [magnetic tape tail extend=0cm]    at (0,0) {A};
  \node [magnetic tape tail extend=0.25cm] at (0,1) {B};
\end{tikzpicture}
```

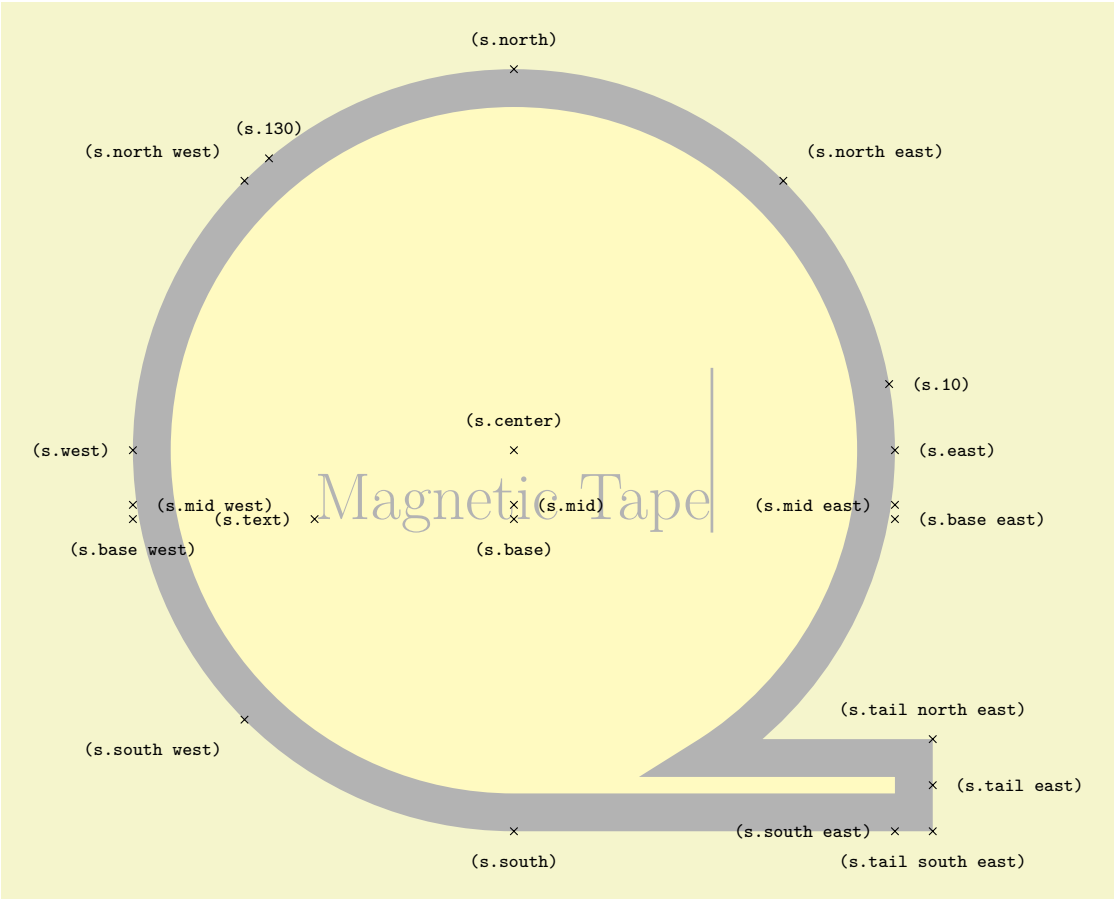
/pgf/magnetic tape tail=⟨*proportion*⟩ (no default, initially `0.15`)

This key sets the thickness of the 'tail' to be ⟨*proportion*⟩ times the radius of the shape. The ⟨*proportion*⟩ should be between `0.0` and `1.0`.




```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}[every node/.style={magnetic tape, draw}]
  \node [magnetic tape tail=0.5, magnetic tape tail extend=0.5cm] {A};
  \node [magnetic tape tail=0.25] at (0,1) {B};
\end{tikzpicture}
```

The following figure shows the anchors this shape defines; the anchors 10 and 130 are example of border anchors.



```
\usetikzlibrary {shapes.symbols}
\Huge
\begin{tikzpicture}
\node[name=s,shape=magnetic tape,shape example,inner sep=0.75cm,
magnetic tape tail extend=0.5cm]
  {Magnetic Tape\vrule width 1pt height 2cm};
\foreach \anchor/\placement in
  {north west/above left, north/above, north east/above right,
  west/left, center/above, east/right,
  mid west/right, mid/right, mid east/left,
  base west/below, base/below, base east/right,
  south west/below left, south/below, south east/left,
  text/left, 10/right, 130/above,
  tail east/right, tail south east/below, tail north east/above}
    \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
      node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.5 Arrow Shapes

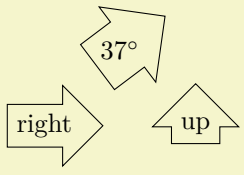
TikZ Library shapes.arrows

```
\usepgflibrary{shapes.arrows}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.arrows]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.arrows} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.arrows] % ConTeXt when using TikZ
```

This library defines arrow shapes. Note that an arrow shape is something quite different from a (normal) arrow tip: It is a shape that just "happens" to "look like" an arrow. In particular, you cannot use these shapes as arrow tips.

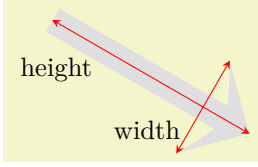
Shape single arrow

This shape is an arrow, which tightly fits the node contents (including any `inner sep`). This shape supports the rotation of the shape border, as described in Section 17.2.3. The angle of rotation determines in which direction the arrow points (provided no other rotational transformations are applied).




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={single arrow, draw},
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
  \node {right};
  \node at (2,0) [shape border rotate=90]{up};
  \node at (1,1) [rotate border=37, inner sep=0pt]{$37^\circ$};
\end{tikzpicture}
```

Regardless of the rotation of the arrow border, the width is measured between the back ends of the arrow head, and the height is measured from the arrow tip to the end of the arrow tail.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[>=stealth,
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
  \node[rotate border=-30, fill=gray!25, minimum height=3cm, single arrow,
    single arrow head extend=.5cm, single arrow head indent=.25cm] (arrow) {};
  \draw[red, <->] (arrow.before tip) -- (arrow.after tip)
    node [near end, left, black] {width};
  \draw[red, <->] (arrow.tip) -- (arrow.tail)
    node [near end, below left, black] {height};
\end{tikzpicture}
```

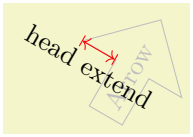
There are PGF keys that can be used to customize this shape (to use these keys in TikZ, simply remove the /pgf/ path).

`/pgf/single arrow tip angle=`⟨*angle*⟩ (no default, initially `90`)

Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

`/pgf/single arrow head extend=`⟨*length*⟩ (no default, initially `.5cm`)

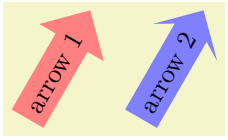
This sets the distance between the tail of the arrow and the outer end of the arrow head. This may change if the shape is enlarged to some minimum width.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
  \node[single arrow, draw, single arrow head extend=.5cm, gray!50, rotate=60]
     (a) {Arrow};
  \draw[red, |<->|] (a.before tip) -- (a.before head)
    node [midway, below, sloped, black] {head extend};
\end{tikzpicture}
```

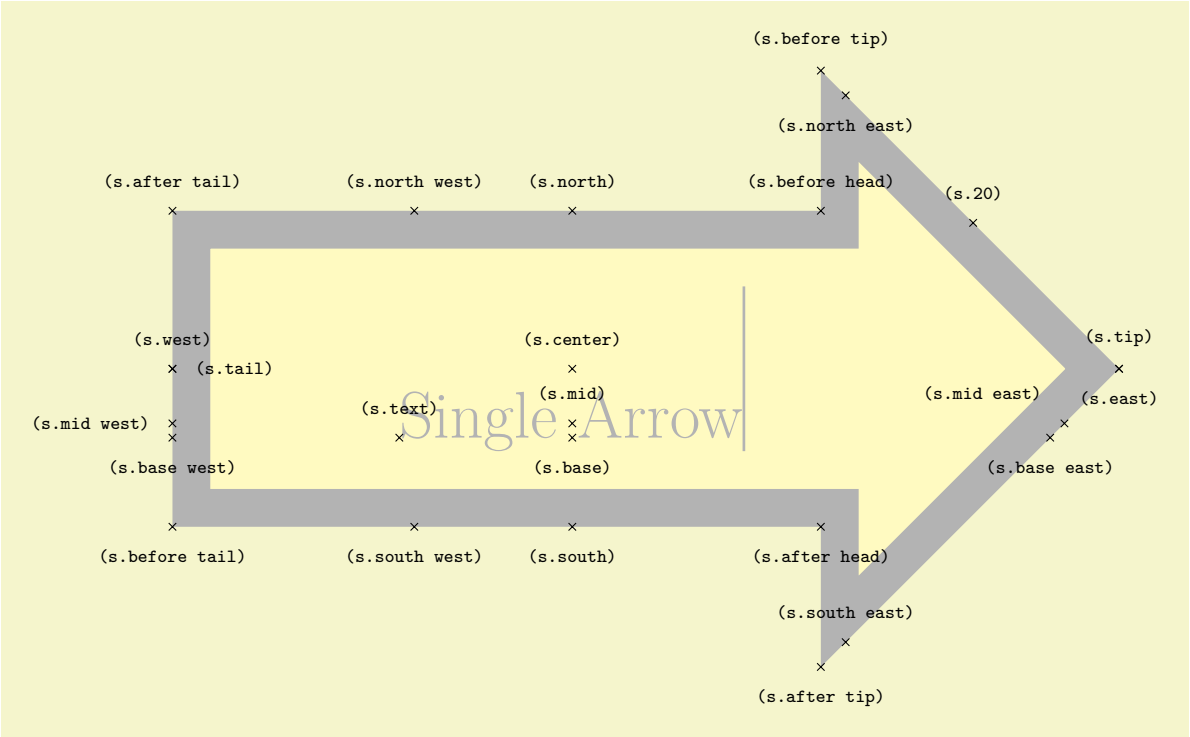
/pgf/single arrow head indent=⟨*length*⟩ (no default, initially 0cm)

This moves the point where the arrow head joins the shaft of the arrow *towards* the arrow tip, by ⟨*length*⟩.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={single arrow, draw=none, rotate=60}]
  \node [fill=red!50]                                          {arrow 1};
  \node [fill=blue!50, single arrow head indent=1ex] at (1.5,0) {arrow 2};
\end{tikzpicture}
```

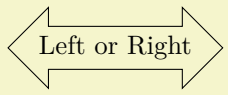
The anchors for this shape are shown below (anchor 20 is an example of a border anchor).




```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=single arrow, shape example, single arrow head extend=1.5cm]
    {Single Arrow\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {text/above,       center/above, 20/above,
     mid west/left,    mid/above,    mid east/above left,
     base west/below, base/below,    base east/below,
     tip/above, before tip/above, after tip/below, before head/above,
     after head/below, after tail/above, before tail/below, tail/right,
     north/above, south/below, east/below, west/above,
     north west/above, north east/below, south west/below, south east/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

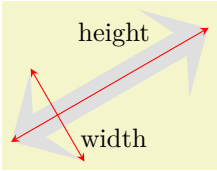
Shape double arrow

This shape is a double arrow, which tightly fits the node contents (including any `inner sep`), and supports the rotation of the shape border, as described in Section 17.2.3.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={double arrow, draw}]
  \node [double arrow, draw] {Left or Right};
\end{tikzpicture}
```

The double arrow behaves exactly like the single arrow, so you need to remember that the width is *always* the distance between the back ends of the arrow heads, and the height is *always* the tip-to-tip distance.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[>=stealth,
    rotate border/.style={shape border uses incircle, shape border rotate=#1}]
  \node[rotate border=210, fill=gray!25, minimum height=3cm, double arrow,
    double arrow head extend=.5cm, double arrow head indent=.25cm] (arrow) {};
  \draw[red, <->] (arrow.before tip 1) -- (arrow.after tip 1)
    node [near start, right, black] {width};
  \draw[red, <->] (arrow.tip 1) -- (arrow.tip 2)
    node [near end, above left, black] {height};
\end{tikzpicture}
```

The PGF keys that can be used to customize the double arrow behave similarly to the keys for the single arrow (to use these keys in Ti*k*Z, simply remove the `/pgf/` path).

`/pgf/double arrow tip angle=`⟨*angle*⟩ (no default, initially `90`)

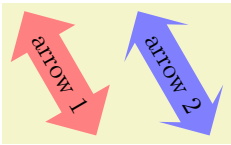
Sets the angle for the arrow tip. Enlarging the arrow to some minimum width may increase the height of the shape to maintain this angle.

`/pgf/double arrow head extend=`⟨*length*⟩ (no default, initially `.5cm`)

This sets the distance between the shaft of the arrow and the outer end of the arrow heads. This may change if the shape is enlarged to some minimum width.

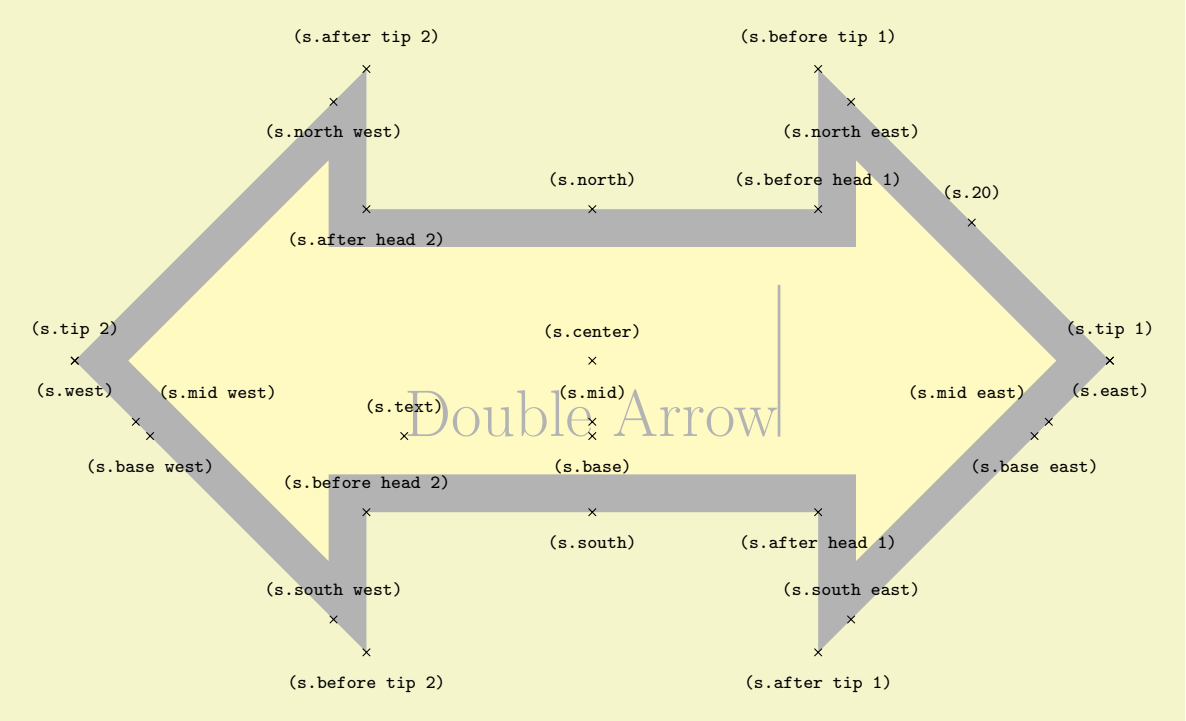
`/pgf/double arrow head indent=`⟨*length*⟩ (no default, initially `0cm`)

This moves the point where the arrow heads join the shaft of the arrow *towards* the arrow tips, by ⟨*length*⟩.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}[every node/.style={double arrow, draw=none, rotate=-60}]
  \node [fill=red!50]                                          {arrow 1};
  \node [fill=blue!50, double arrow head indent=1ex] at (1.5,0) {arrow 2};
\end{tikzpicture}
```

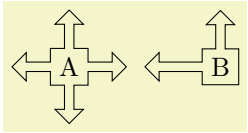
The anchors for this shape are shown below (anchor `20` is an example of a border anchor).




```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=double arrow, double arrow head extend=1.5cm, shape example, inner xsep=2cm]
    {Double Arrow\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {text/above, center/above, 20/above,
     mid west/above right, mid/above, mid east/above left,
     base west/below, base/below, base east/below,
     before head 1/above, before tip 1/above, tip 1/above, after tip 1/below, after head 1/below,
     before head 2/above, before tip 2/below, tip 2/above, after tip 2/above, after head 2/below,
     north/above, south/below, east/below, west/below,
     north west/below, north east/below, south west/above, south east/above}
     \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

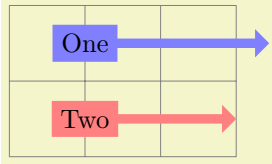
Shape `arrow box`

This shape is a rectangle with optional arrows which extend from the four sides.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
  \node[arrow box, draw] {A};
  \node[arrow box, draw, arrow box arrows={north:.5cm, west:0.75cm}]
    at (2,0) {B};
\end{tikzpicture}
```

Any minimum size requirements are applied to the main rectangle *only*. This does not pose too many problems if you wish to accommodate the length of the arrows, as it is possible to specify the length of each arrow independently, from either the border of the rectangle (the default) or the center of the rectangle.




```
\usetikzlibrary {shapes.arrows}
\begin{tikzpicture}
    \tikzset{box/.style={arrow box, fill=#1}}
    \draw [help lines] grid(3,2);
  \node[box=blue!50, arrow box arrows={east:2cm}]                 at (1,1.5){One};
  \node[box=red!50,  arrow box arrows={east:2cm from center}] at (1,0.5){Two};
\end{tikzpicture}
```

There are various PGF keys for drawing this shape (to use these keys in Ti*k*Z, simply remove the /pgf/ path).

`/pgf/arrow box tip angle=`⟨*angle*⟩ (no default, initially `90`)

Sets the angle at the arrow tip for all four arrows.

`/pgf/arrow box head extend=`⟨*length*⟩ (no default, initially `.125cm`)

Sets the distance the arrow head extends away from the shaft of the arrow. This applies to all arrows.

`/pgf/arrow box head indent=`⟨*length*⟩ (no default, initially `0cm`)

Moves the point where the arrow head joins the shaft of the arrow *towards* the arrow tip. This applies to all arrows.

`/pgf/arrow box shaft width=`⟨*length*⟩ (no default, initially `.125cm`)

Sets the width of the shaft of all arrows.

`/pgf/arrow box north arrow=`⟨*distance*⟩ (no default, initially `.5cm`)

Sets the distance the north arrow extends from the node. By default this is from the border of the shape, but by using the additional keyword `from center`, the distance will be measured from the center of the shape. If ⟨*distance*⟩ is `0pt` or a negative distance, the arrow will not be drawn.

`/pgf/arrow box south arrow=`⟨*distance*⟩ (no default, initially `.5cm`)

Sets the distance the south arrow extends from the node.

`/pgf/arrow box east arrow=`⟨*distance*⟩ (no default, initially `.5cm`)

Sets the distance the east arrow extends from the node.

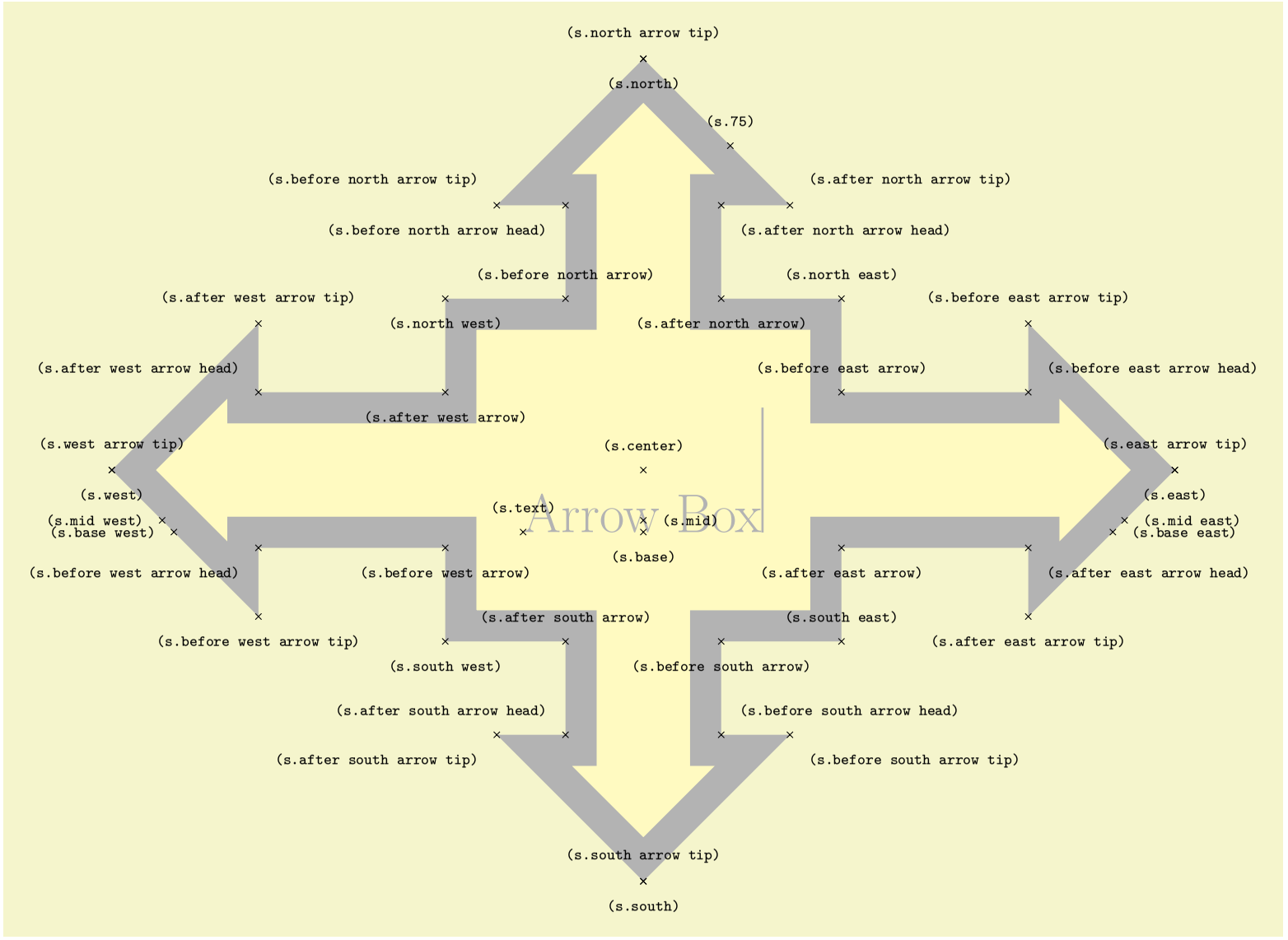
`/pgf/arrow box west arrow=`⟨*distance*⟩ (no default, initially `.5cm`)

Sets the distance the west arrow extends from the node.

`/pgf/arrow box arrows={`⟨*list*⟩`}` (no default)

Sets the distance that all arrows extend from the node. The specification in ⟨*list*⟩ consists of the four compass points `north`, `south`, `east` or `west`, separated by commas (so the list must be contained within braces). The distances can be specified after each side separated by a colon (e.g., `north:1cm`, or `west:5cm from center`). If an item specifies no distance, the most recently specified distance will be used (at the start of the list this is `0cm`, so the first item in the list should specify a distance). Any sides not specified will not be drawn with an arrow.

The anchors for this shape are shown below (unfortunately, due to its size, this example must be rotated). Anchor `75` is an example of a border anchor. If a side is drawn without an arrow, the anchors for that arrow should be considered unavailable. They are (unavoidably) defined, but default to the center of the appropriate side.


(s.north arrow tip)
(s.north)
(s.75)
(s.before north arrow tip)
(s.after north arrow tip)
(s.before north arrow head)
(s.after north arrow head)
(s.before north arrow)
(s.north east)
(s.after west arrow tip)
(s.north west)
(s.after north arrow)
(s.before east arrow tip)
(s.after west arrow head)
(s.before east arrow)
(s.before east arrow head)
(s.after west arrow)
(s.west arrow tip)
(s.center)
(s.east arrow tip)
(s.west)
(s.east)
(s.text)
(s.mid)
(s.mid west)
(s.mid east)
(s.base west)
(s.base east)
(s.base)
(s.before west arrow head)
(s.before west arrow)
(s.after east arrow)
(s.after east arrow head)
(s.after south arrow)
(s.south east)
(s.before west arrow tip)
(s.after east arrow tip)
(s.south west)
(s.before south arrow)
(s.after south arrow head)
(s.before south arrow head)
(s.after south arrow tip)
(s.before south arrow tip)
(s.south arrow tip)
(s.south)
Arrow Box

```
\usetikzlibrary {shapes.arrows}
\Huge
\begin{tikzpicture}
  \node[shape=arrow box, shape example, inner xsep=1cm, inner ysep=1.5cm, arrow box shaft width=2cm,
    arrow box arrows={north:3.5cm from border, south, east:5cm from border, west},
    arrow box head extend=0.75cm, rotate=-90](s) {Arrow Box\vrule width1pt height2cm};
  \foreach \anchor/\placement in
   {center/above, text/above, mid/right, base/below, 75/above,
    mid east/right, mid west/left, base east/right, base west/left,
    north/below, south/below, east/below, west/below,
    north east/above, south east/above, south west/below, north west/below,
    north arrow tip/above,south arrow tip/above, east arrow tip/above, west arrow tip/above,
    before north arrow/above, before north arrow head/below left, before north arrow tip/above left,
    after north arrow tip/above right, after north arrow head/below right, after north arrow/below,
    before south arrow/below, before south arrow head/above right, before south arrow tip/below right,
    after south arrow tip/below left, after south arrow head/above left, after south arrow/above,
    before east arrow/above, before east arrow head/above right, before east arrow tip/above,
    after east arrow tip/below, after east arrow head/below right, after east arrow/below,
    before west arrow/below, before west arrow head/below left, before west arrow tip/below,
    after west arrow tip/above, after west arrow head/above left, after west arrow/below}
      \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
        node[\placement, rotate=-90] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.6 Shapes with Multiple Text Parts

TikZ Library shapes.multipart

```
\usepgflibrary{shapes.multipart}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.multipart]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.multipart} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.multipart] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that are composed of multiple (text) parts.

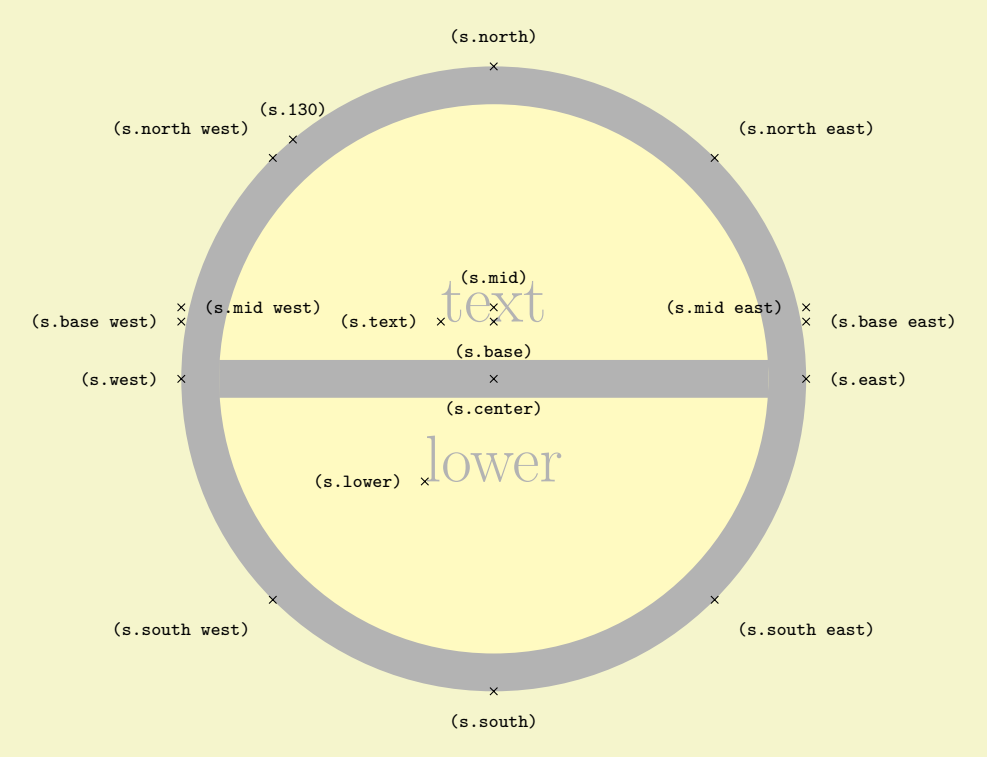
Shape circle split

This shape is a multi-part shape consisting of a circle with a line in the middle. The upper part is the main part (the `text` part), the lower part is the `lower` part.




```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
  \node [circle split,draw,double,fill=red!20]
  {
    $q_1$
    \nodepart{lower}
    $00$
  };
\end{tikzpicture}
```

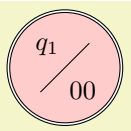
The shape inherits all anchors from the `circle` shape and defines the `lower` anchor in addition. See also the following figure:



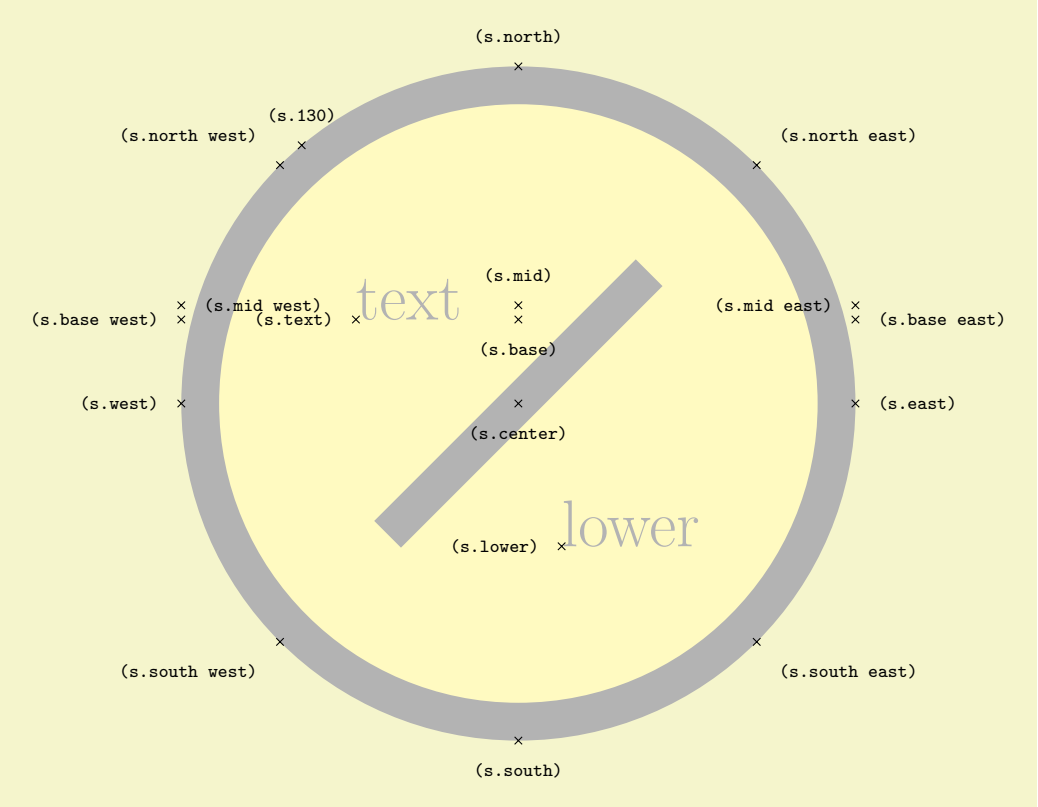

```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle split,shape example] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/below, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, lower/left, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `circle solidus`

This shape (due to Manuel Lacruz) is similar to the split circle, but the two text parts are arranged diagonally.



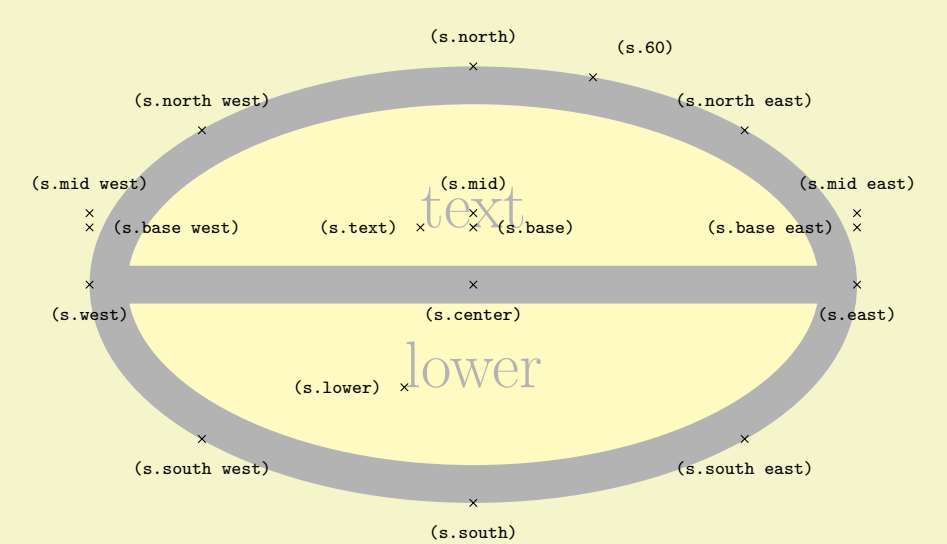

```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
  \node [circle solidus,draw,double,fill=red!20]
  {
    $q_1$
    \nodepart{lower}
    $00$
  };
\end{tikzpicture}
```




```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=circle solidus,shape example,inner xsep=1cm] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/below, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, lower/left, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `ellipse split`

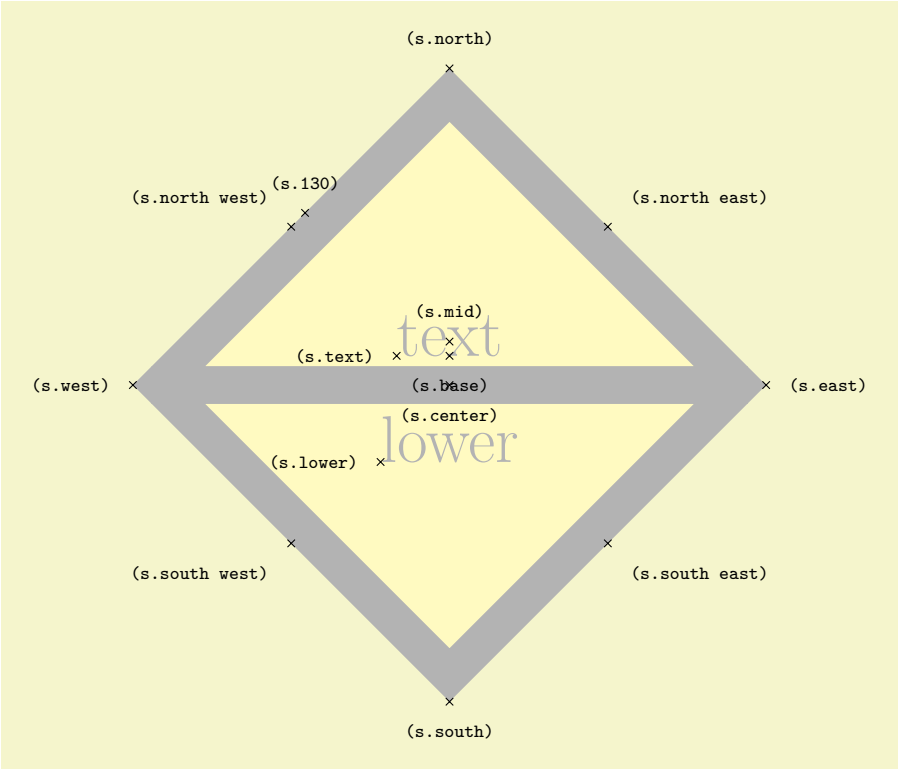
This shape is a multi-part shape consisting of an ellipse with a line in the middle. The upper part is the main part (the `text` part), the lower part is the `lower` part. The anchors for this shape are shown below. Anchor `60` is a border anchor.




```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse split,shape example] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {center/below, text/left, lower/left, 60/above right,
     mid/above, mid east/above, mid west/above,
     base/right, base east/left, base west/right,
     north/above, south/below, east/below, west/below,
     north east/above, south east/below, south west/below, north west/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `diamond split`

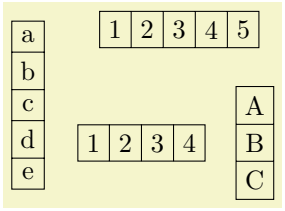
This shape is a multi-part shape consisting of a diamond with a line in the middle. The upper part is the main part (the `text` part); the lower part is the `lower` part.




```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=diamond split,shape example,inner xsep=1cm] {text\nodepart{lower}lower};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/below, east/right, mid/above, base/below,
     south west/below left, south/below, south east/below right,
     text/left, lower/left, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `rectangle split`

This shape is a rectangle, which can be split either horizontally or vertically into several parts.




```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}[my shape/.style={
  rectangle split, rectangle split parts=#1, draw, anchor=center}]
  \node [my shape=5] at (0,1)
    {a\nodepart{two}b\nodepart{three}c\nodepart{four}d\nodepart{five}e};
  \node [my shape=5, rectangle split horizontal] at (2,2)
    {1\nodepart{two}2\nodepart{three}3\nodepart{four}4\nodepart{five}5};
  \node [my shape=3] at (3,0.5)
    {A\nodepart{two}B\nodepart{three}C};
  \node [my shape=4, rectangle split horizontal] at (1.5,0.5)
    {1\nodepart{two}2\nodepart{three}3\nodepart{four}4};
\end{tikzpicture}
```

The shape can be split into a maximum of twenty parts. However, to avoid allocating a lot of unnecessary boxes, PGF only allocates four boxes by default. To use the `rectangle split` shape with more than four boxes, the extra boxes must be allocated manually in advance (perhaps using `\newbox` or `\let`). The boxes take the form `\pgfnodepart`⟨*number*⟩`box`, where ⟨*number*⟩ is from the cardinal numbers `one`, `two`, `three`, … and so on. `\pgfnodepartonebox` is special in that it is synonymous with `\pgfnodeparttextbox`. For compatibility with earlier versions of this shape, the boxes `\pgfnodeparttwobox`, `\pgfnodepartthreebox` and `\pgfnodepartfourbox`, can be referred to using the ordinal numbers: `\pgfnodepartsecondbox`, `\pgfnodepartthirdbox` and `\pgfnodepartfourthbox`. In order to facilitate the allocation of these extra boxes, the following key is provided:

`/pgf/rectangle split allocate boxes=`⟨*number*⟩ (no default)

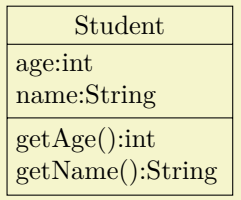
This key checks if ⟨*number*⟩ boxes have been allocated, and if not, it allocates the required boxes using `\newbox` (some "magic" is performed to get around the fact that `\newbox` is declared `\outer` in plain TEX).

When split vertically, the rectangle split will meet any `minimum width` requirements, but any `minimum height` will be ignored. Conversely when split horizontally, `minimum height` requirements will be met, but any `minimum width` will be ignored. In addition, `inner sep` is applied to every part that is used, so it cannot be specified independently for a particular part.

There are several PGF keys to specify how the shape is drawn. To use these keys in Ti*k*Z, simply remove the `/pgf/` path:

`/pgf/rectangle split parts=`⟨*number*⟩ (no default, initially `4`)

Split the rectangle into ⟨*number*⟩ parts, which should be in the range `1` to `20`. If more than four parts are needed, the boxes should be allocated in advance as described above.



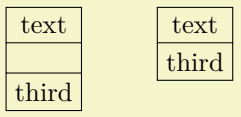

```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}[every text node part/.style={align=center}]
  \node[rectangle split, rectangle split parts=3, draw, text width=2.75cm]
    {Student
     \nodepart{two}
       age:int \\
       name:String
     \nodepart{three}
       getAge():int \\
       getName():String};
\end{tikzpicture}
```

`/pgf/rectangle split horizontal=`⟨*boolean*⟩ (default `true`)

This key determines whether the rectangle is split horizontally or vertically

`/pgf/rectangle split ignore empty parts=`⟨*boolean*⟩ (default `true`)

When ⟨*boolean*⟩ is true, PGF will ignore any part that is empty *except the text part*. This effectively overrides the `rectangle split parts` key in that, if 3 parts (for example) are specified, but one is empty, only two will be shown.




```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}[every node/.style={draw, anchor=text, rectangle split,
    rectangle split parts=3}]
  \node {text \nodepart{second} \nodepart{third}third};
  \node [rectangle split ignore empty parts] at (2,0)
        {text \nodepart{second} \nodepart{third}third};
\end{tikzpicture}
```

`/pgf/rectangle split empty part width=`⟨*length*⟩ (no default, initially `1ex`)

Sets the default width for a node part box if it is empty and empty parts are not ignored.

`/pgf/rectangle split empty part height=`⟨*length*⟩ (no default, initially `1ex`)

Sets the default height for a node part box if it is empty and empty parts are not ignored.

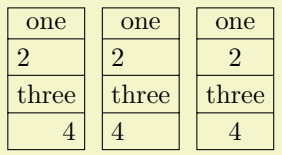
`/pgf/rectangle split empty part depth=`⟨*length*⟩ (no default, initially `0ex`)

Sets the default depth for a node part box if it is empty and empty parts are not ignored.

`/pgf/rectangle split part align={`⟨*list*⟩`}` (no default, initially `center`)

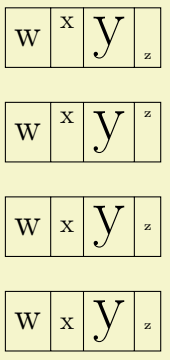
Sets the alignment of the boxes inside the node parts. Each item in ⟨*list*⟩ should be separated by commas (so if there is more than one item in ⟨*list*⟩, it must be surrounded by braces).

When the rectangle is split vertically, the entries in ⟨*list*⟩ must be one of `left`, `right`, or `center`. If ⟨*list*⟩ has less entries than node parts then the remaining boxes are aligned according to the last entry in the list. Note that this only aligns the boxes in each part and *does not* affect the alignment of the contents of the boxes.




```
\usetikzlibrary {shapes.multipart}
\def\x{one \nodepart{two} 2 \nodepart{three} three \nodepart{four} 4}
\begin{tikzpicture}[
  every node/.style={rectangle split, rectangle split parts=4,
    draw}
  ]
  \node[rectangle split part align={center, left, right}] at (0,0)    {\x};
  \node[rectangle split part align={center, left}]        at (1.25,0) {\x};
  \node[rectangle split part align={center}]              at (2.5,0)  {\x};
\end{tikzpicture}
```

When the rectangle is split horizontally, the entries in ⟨*list*⟩ must be one of `top`, `bottom`, `center` or `base`. Note that using the value `base` will only make sense if all the node part boxes are being aligned in this way. This is because the `base` value aligns the boxes in relation to each other, whereas the other values align the boxes in relation to the part of the shape they occupy.




```
\usetikzlibrary {shapes.multipart}
\def\x{\Large w\nodepart{two}x\nodepart{three}\Huge y\nodepart{four}\tiny z}
\begin{tikzpicture}[
  every node/.style={rectangle split, rectangle split parts=4,
    draw, rectangle split horizontal}
  ]
  \node[rectangle split part align={center, top, bottom}] at (0,0)     {\x};
  \node[rectangle split part align={center, top}]         at (0,-1.25) {\x};
  \node[rectangle split part align={center}]              at (0,-2.5)  {\x};
  \node[rectangle split part align=base]                  at (0,-3.75) {\x};
\end{tikzpicture}
```

`/pgf/rectangle split draw splits=`⟨*boolean*⟩ (default `true`)

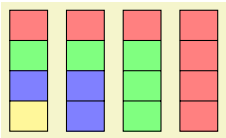
Sets whether the line or lines between node parts will be drawn. Internally, this sets the TEX-if `\ifpgfrectanglesplitdrawsplits` appropriately.

`/pgf/rectangle split use custom fill=`⟨*boolean*⟩ (default `true`)

This enables the use of a custom fill for each of the node parts (including the area covered by the `inner sep`). The background path for the shape should not be filled (e.g., in Ti*k*Z, the `fill` option for the node must be implicitly or explicitly set to `none`). Internally, this key sets the TEX-if `\ifpgfrectanglesplitusecustomfill` appropriately.

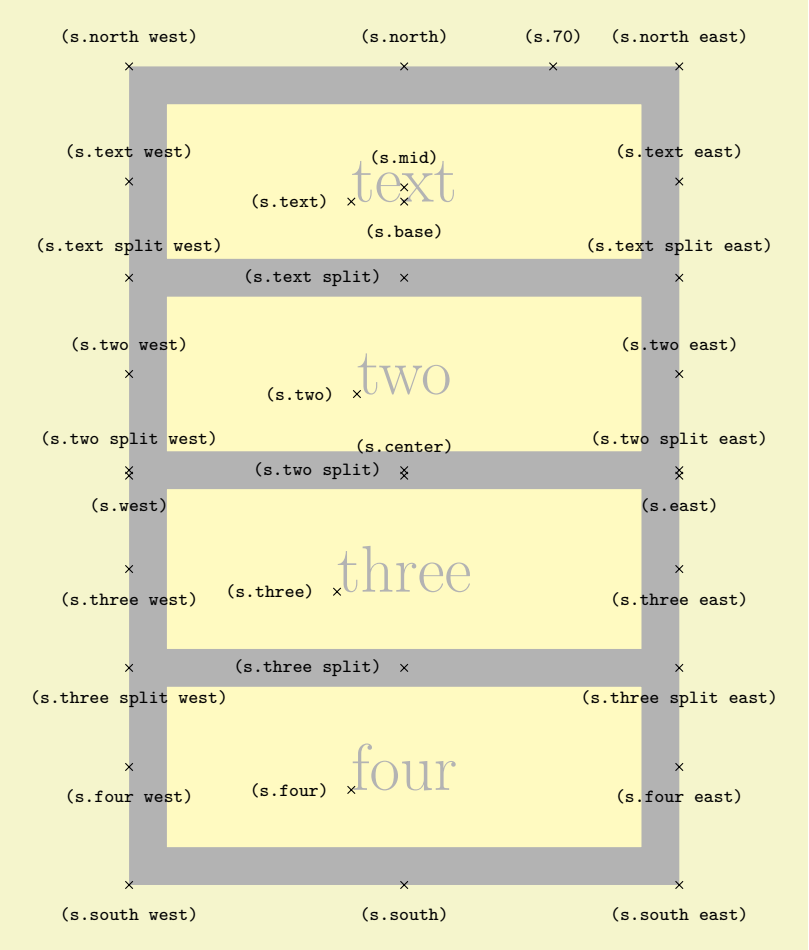
`/pgf/rectangle split part fill={`⟨*list*⟩`}` (no default, initially `white`)

Sets the custom fill color for each node part shape. The items in ⟨*list*⟩ should be separated by commas (so if there is more than one item in ⟨*list*⟩, it must be surrounded by braces). If ⟨*list*⟩ has less entries than node parts, then the remaining node parts use the color from the last entry in the list. This key will automatically set `/pgf/rectangle split use custom fill`.



```
\usetikzlibrary {shapes.multipart}
\begin{tikzpicture}
  \tikzset{every node/.style={rectangle split, draw, minimum width=.5cm}}
  \node[rectangle split part fill={red!50, green!50, blue!50, yellow!50}]  {};
  \node[rectangle split part fill={red!50, green!50, blue!50}] at (0.75,0) {};
  \node[rectangle split part fill={red!50, green!50}]          at (1.5,0)  {};
  \node[rectangle split part fill={red!50}]                    at (2.25,0) {};
\end{tikzpicture}
```

The anchors for the `rectangle split` shape split vertically into four, are shown below (anchor `70` is an example of a border angle). When a node part is missing, the anchors prefixed with the name of that node part should be considered unavailable. They are (unavoidably) defined, but default to other anchor positions.




```
\usetikzlibrary {shapes.multipart}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle split, rectangle split parts=4, shape example,
    inner ysep=0.75cm]
    {\nodepart{text}text\nodepart{two}two
       \nodepart{three}three\nodepart{four}four};
  \foreach \anchor/\placement in
    {text/left, text east/above, text west/above,
     two/left, two east/above, two west/above,
     three/left, three east/below, three west/below,
     four/left, four east/below, four west/below,
     text split/left, text split east/above, text split west/above,
     two split/left, two split east/above, two split west/above,
     three split/left, three split east/below, three split west/below,
     north/above, south/below, east/below, west/below,
     north west/above, north east/above, south west/below, south east/below,
     center/above, 70/above, mid/above, base/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.7 Callout Shapes

TikZ Library shapes.callouts

```
\usepgflibrary{shapes.callouts}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.callouts]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.callouts} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.callouts] % ConTeXt when using TikZ
```

Producing basic callouts can be done quite easily in PGF and TikZ by creating a node and then subse-

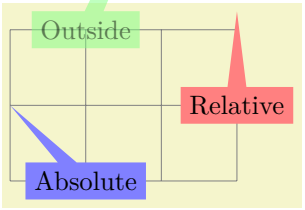
quently drawing a path from the border of the node to the required point. This library provides more fancy, 'balloon'-style callouts.

Callouts consist of a main shape and a pointer (which is part of the shape) which points to something in (or outside) the picture. The position on the border of the main shape to which the pointer is connected is determined automatically. However, the pointer is ignored when calculating the minimum size of the shape, and also when positioning anchors.




```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}[remember picture]
  \node[ellipse callout, draw] (hallo) {Hallo!};
\end{tikzpicture}
```

There are two kinds of pointer: the "relative" pointer and the "absolute" pointer. The relative pointer calculates the angle of a specified coordinate relative to the center of the main shape, locates the point on the border to which this angle corresponds, and then adds the coordinate to this point. This seemingly over-complex approach means than you do not have to guess the size of the main shape: the relative pointer will always be outside. The absolute pointer, on the other hand, is much simpler: it points to the specified coordinate absolutely (and can even point to named coordinates in different pictures).




```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}[remember picture, note/.style={rectangle callout, fill=#1}]
  \draw [help lines] grid(3,2);
  \node [note=red!50,  callout relative pointer={(0,1)}] at (3,1) {Relative};
  \node [note=blue!50, callout absolute pointer={(0,1)}] at (1,0) {Absolute};
  \node [note=green!50, opacity=.5, overlay,
         callout absolute pointer={(hallo.south)}]       at (1,2) {Outside};
\end{tikzpicture}
```

The following keys are common to all callouts. Please remember that the `callout relative pointer`, and `callout absolute pointer` keys take a different format for their value depending on whether they are being used in PGF or TikZ.

`/pgf/callout relative pointer=`⟨*coordinate*⟩ (no default, initially `\pgfpointpolar{315}{.5cm}`)

Sets the vector of the callout pointer 'relative' to the callout shape.

`/pgf/callout absolute pointer=`⟨*coordinate*⟩ (no default)

Sets the vector of the callout pointer absolutely within the picture.

`/tikz/callout relative pointer=`⟨*coordinate*⟩ (no default, initially `(315:.5cm)`)

The TikZ version of the `callout relative pointer` key. Here, ⟨*coordinate*⟩ can be specified using the TikZ format for coordinates.

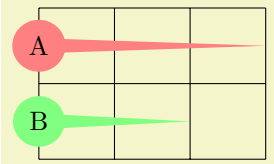
`/tikz/callout absolute pointer=`⟨*coordinate*⟩ (no default)

The TikZ version of the `callout absolute pointer` key. Here, ⟨*coordinate*⟩ can be specified using the TikZ format for coordinates.

It is also possible to shorten the pointer by some distance, using the following key:

`/pgf/callout pointer shorten=`⟨*distance*⟩ (no default, initially `0cm`)

Moves the callout pointer towards the center of the callout's main shape by ⟨*distance*⟩.




```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}
    \tikzset{callout/.style={ellipse callout, callout pointer arc=30,
      callout absolute pointer={#1}}}
  \draw (0,0) grid (3,2);
  \node[callout={(3,1.5)}, fill=red!50] at (0,1.5) {A};
  \node[callout={(3,.5)},  fill=green!50, callout pointer shorten=1cm]
    at (0,.5)  {B};
\end{tikzpicture}
```

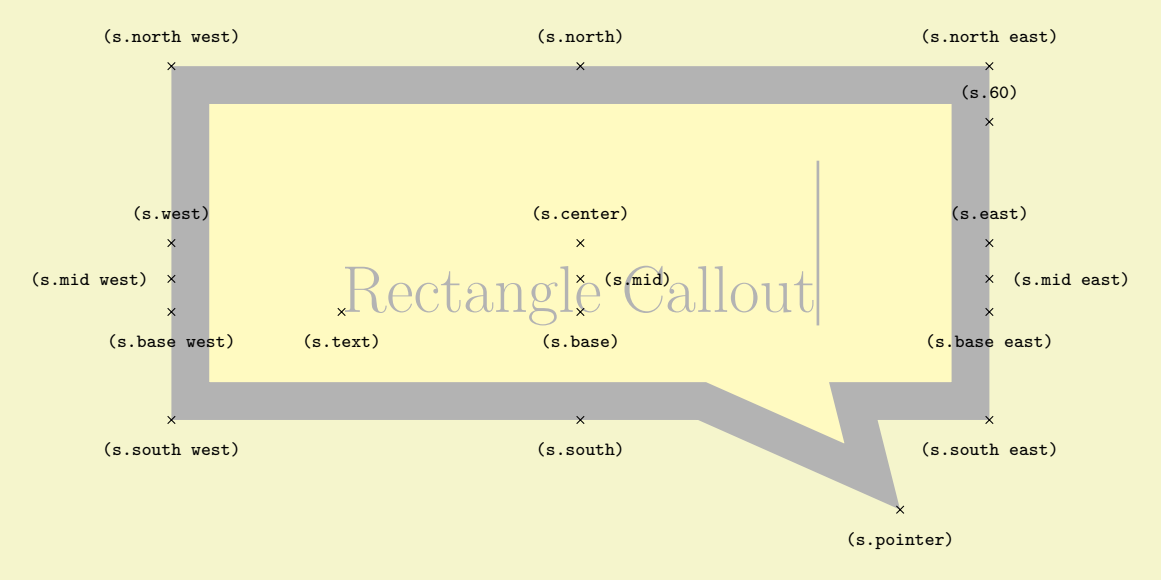
Shape rectangle callout

This shape is a callout whose main shape is a rectangle, which tightly fits the node contents (including any `inner sep`). It supports the keys described above and also the following key:

`/pgf/callout pointer width=`⟨*length*⟩ (no default, initially `.25cm`)

Sets the width of the pointer at the border of the rectangle.

The anchors for this shape are shown below (anchor `60` is an example of a border anchor). The pointer direction is ignored when placing anchors. Additionally, when using an absolute pointer, the `pointer` anchor should not to be used to used to position the shape as it is calculated whilst the shape is being drawn.




```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rectangle callout, callout relative pointer={(1.25cm,-1cm)},
     callout pointer width=2cm, shape example, inner xsep=2cm, inner ysep=1cm]
      {Rectangle Callout\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {center/above, text/below,      60/above,
     mid/right,    mid west/left,   mid east/right,
     base/below,   base west/below, base east/below,
     north/above,  south/below, east/above, west/above,
     north west/above, north east/above,
     south west/below, south east/below,
     pointer/below}
     \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

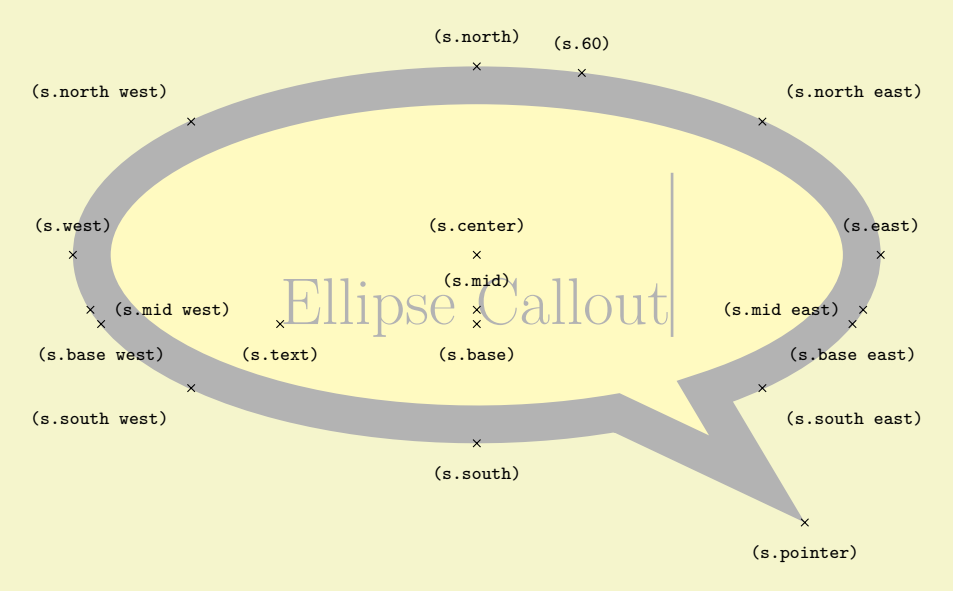
Shape ellipse callout

This shape is a callout whose main shape is an ellipse, which tightly fits the node contents (including any `inner sep`). It uses the `absolute callout pointer`, `relative callout pointer` and `callout pointer shorten` keys, and also the following key:

`/pgf/callout pointer arc=`⟨*angle*⟩ (no default, initially `15`)

Sets the width of the pointer at the border of the ellipse according to an arc of length ⟨*angle*⟩.

The anchors for this shape are shown below (anchor `60` is an example of a border anchor). The pointer direction is ignored when placing anchors and the `pointer` anchor can only be used to position the shape when the relative anchor is specified.




```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=ellipse callout, callout relative pointer={(1.25cm,-1cm)},
    callout pointer width=2cm, shape example, inner xsep=1cm, inner ysep=.5cm]
      {Ellipse Callout\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {center/above, text/below,      60/above,
     mid/above,    mid west/right,  mid east/left,
     base/below,   base west/below, base east/below,
     north/above,  south/below, east/above, west/above,
     north west/above left,    north east/above right,
     south west/below left,    south east/below right,
     pointer/below}
     \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `cloud callout`

This shape is a callout whose main shape is a cloud which fits the node contents. The pointer is segmented, consisting of a series of shrinking ellipses. This callout requires the `shapes.callouts` library (for the cloud shape). If this library is not loaded an error will result.




```
\usetikzlibrary {shapes.callouts}
\begin{tikzpicture}
  \node[cloud callout, cloud puffs=15, aspect=2.5, cloud puff arc=120,
    shading=ball,text=white] {\bf Imagine...};
\end{tikzpicture}
```

The `cloud callout` supports the `absolute callout pointer`, `relative callout pointer` and `callout pointer shorten` keys, as described above. The main shape can be modified using the same keys as the `cloud` shape. The following keys are also supported:

`/pgf/callout pointer start size=`⟨*value*⟩ (no default, initially `.2 of callout`)

Sets the size of the first segment in the pointer (i.e., the segment nearest the main cloud shape). There are three possible forms for ⟨*value*⟩:

- A single dimension (e.g., `5pt`), in which case the first ellipse will have equal diameters of 5pt.
- Two dimensions (e.g., `10pt and 2.5pt`), which sets the x and y diameters of the first ellipse.
- A decimal fraction (e.g., `.2 of callout`), in which case the x and y diameters of the first ellipse will be set as fractions of the width and height of the main shape. The keyword `of callout` cannot be omitted.

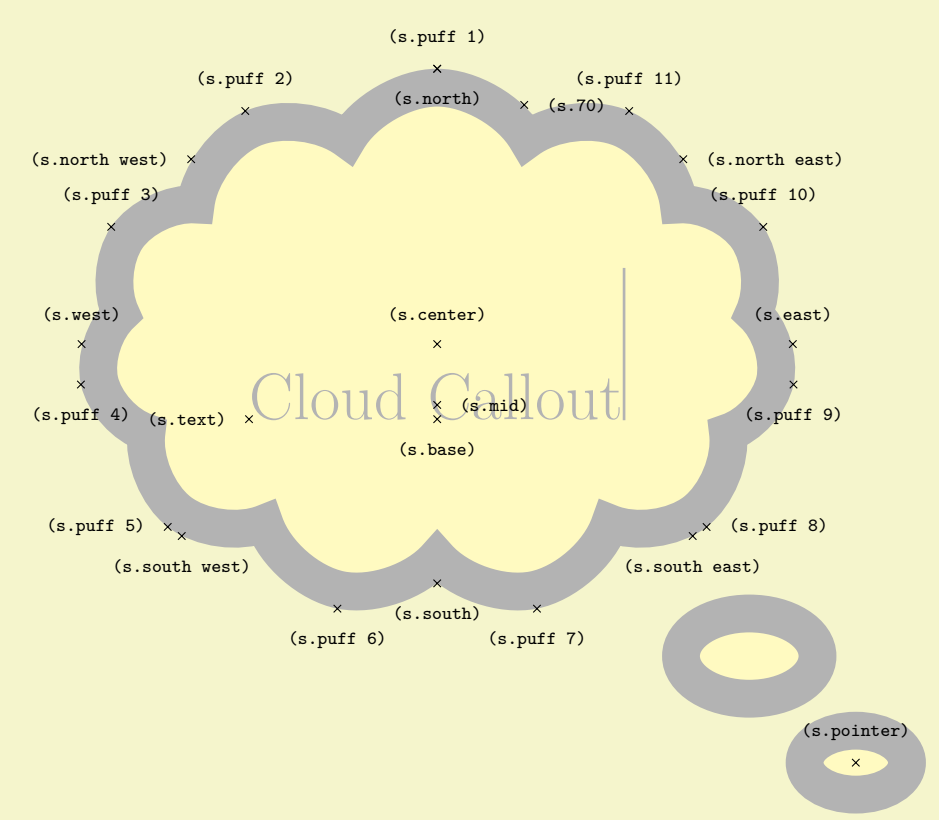
/pgf/callout pointer end size=⟨*value*⟩ (no default, initially `.1 of callout`)

Sets the size of the last ellipse in the pointer.

/pgf/callout pointer segments=⟨*number*⟩ (no default, initially `2`)

Sets the number of segments in the pointer. Note that PGF will happily overlap segments if too many are specified.

The anchors for this shape are shown below (anchor `70` is an example of a border anchor). The pointer direction is ignored when placing anchors and the pointer anchor can only be used to position the shape when the relative anchor is specified. Note that the center of the last segment is drawn at the `pointer` anchor.




```
\usetikzlibrary {shapes.callouts}
\Huge
\begin{tikzpicture}
  \node[name=s, shape=cloud callout, style=shape example, cloud puffs=11, aspect=1.5,
    cloud puff arc=120,inner xsep=.5cm, callout pointer start size=.25 of callout,
    callout pointer end size=.15 of callout, callout relative pointer={(315:4cm)},
    callout pointer segments=2] {Cloud Callout\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {puff 1/above, puff 2/above, puff 3/above, puff 4/below,
     puff 5/left, puff 6/below, puff 7/below, puff 8/right,
     puff 9/below, puff 10/above, puff 11/above, 70/right,
     center/above, base/below, mid/right, text/left,
     north/below, south/below, east/above, west/above,
     north west/left, north east/right,
     south west/below, south east/below,pointer/above}
  \draw[shift={(s.\anchor)}] plot[mark=x] coordinates{(0,0)}
    node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

71.8 Miscellaneous Shapes

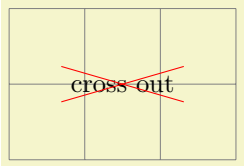
TikZ Library shapes.misc

```
\usepgflibrary{shapes.misc}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[shapes.misc]  % ConTeXt when using pure PGF
\usetikzlibrary{shapes.misc} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[shapes.misc] % ConTeXt when using TikZ
```

This library defines general-purpose shapes that do not fit into the previous categories.

Shape `cross out`

This shape "crosses out" the node. Its foreground path are simply two diagonal lines between the corners of the node's bounding box. Here is an example:



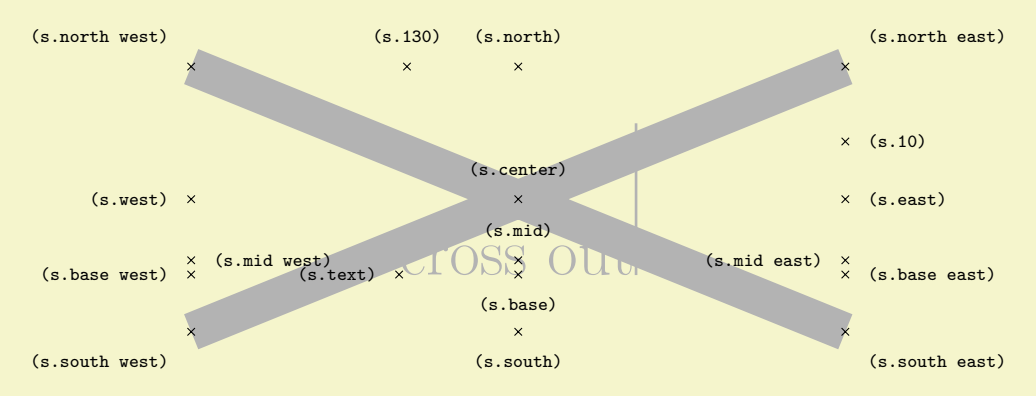

```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \node [cross out,draw=red] at (1.5,1) {cross out};
\end{tikzpicture}
```

A useful application is inside text as in the following example:

Cross ~~me~~ out!

```
\usetikzlibrary {shapes.misc}
Cross \tikz[baseline] \node [cross out,draw,anchor=text] {me}; out!
```

This shape inherits all anchors from the `rectangle` shape, see also the following figure:




```
\usetikzlibrary {shapes.misc}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=cross out,shape example] {cross out\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {north west/above left, north/above, north east/above right,
     west/left, center/above, east/right,
     mid west/right, mid/above, mid east/left,
     base west/left, base/below, base east/right,
     south west/below left, south/below, south east/below right,
     text/left, 10/right, 130/above}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape `strike out`

This shape is identical to the `cross out` shape, only its foreground path consists of a single line from the lower left to the upper right.

Strike ~~me~~ out!

```
\usetikzlibrary {shapes.misc}
Strike \tikz[baseline] \node [strike out,draw,anchor=text] {me}; out!
```

See the `cross out` shape for the anchors.

Shape rounded rectangle

This shape is a rectangle which can have optionally rounded sides.

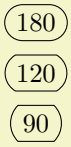



```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \node[rounded rectangle, draw, fill=red!20]{Hallo};
\end{tikzpicture}
```

There are keys to specify how the sides are rounded (to use these keys in Ti*k*Z, simply remove the /pgf/ path).

/pgf/rounded rectangle arc length=⟨*angle*⟩ (no default, initially 180)

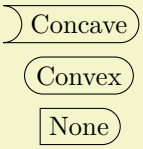
Sets the length of the arcs for the rounded ends. Recommended values for ⟨*angle*⟩ are between 90 and 180.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \matrix[row sep=5pt, every node/.style={draw, rounded rectangle}]{
    \node[rounded rectangle arc length=180] {180}; \\
    \node[rounded rectangle arc length=120] {120}; \\
    \node[rounded rectangle arc length=90]  {90};  \\};
\end{tikzpicture}
```

/pgf/rounded rectangle west arc=⟨*arc type*⟩ (no default, initially convex)

Sets the style of the rounding for the left side. The permitted values for ⟨*arc type*⟩ are concave, convex, or none.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \matrix[row sep=5pt, every node/.style={draw, rounded rectangle}]{
    \node[rounded rectangle west arc=concave] {Concave}; \\
    \node[rounded rectangle west arc=convex]  {Convex};  \\
    \node[rounded rectangle left arc=none]    {None};    \\};
\end{tikzpicture}
```

/pgf/rounded rectangle left arc=⟨*arc type*⟩ (style, no default)

Alternative key for specifying the west arc.

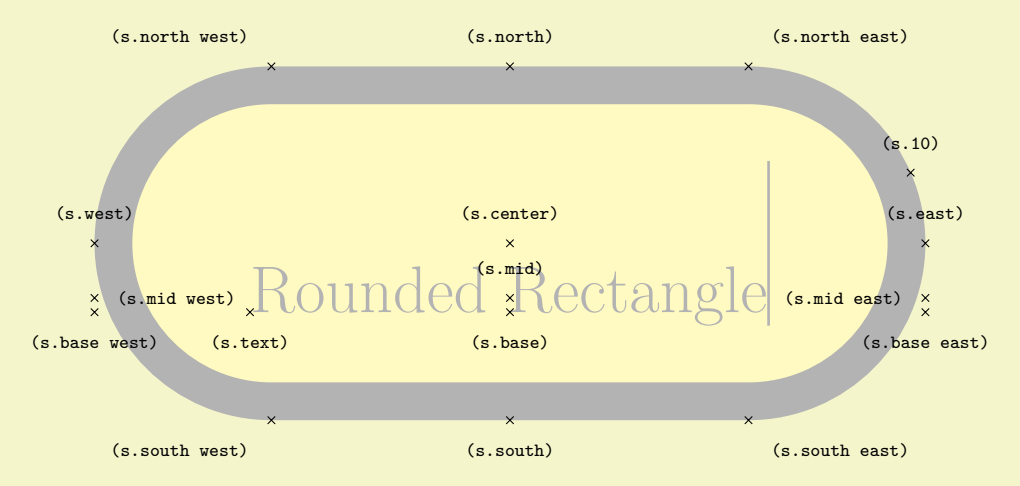
/pgf/rounded rectangle east arc=⟨*arc type*⟩ (no default, initially convex)

Sets the style of the rounding for the east side.

/pgf/rounded rectangle right arc=⟨*arc type*⟩ (style, no default)

Alternative key for specifying the east arc.

The anchors for this shape are shown below (anchor 10 is an example of a border angle). Note that if only one side is rounded, the center anchor will not be the precise center of the shape.




```
\usetikzlibrary {shapes.misc}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=rounded rectangle, shape example, inner xsep=1.5cm, inner ysep=1cm]
      {Rounded Rectangle\vrule width 1pt height 2cm};
  \foreach \anchor/\placement in
    {center/above, text/below,      10/above,
     mid/above,    mid west/right,  mid east/left,
     base/below,   base west/below, base east/below,
     north/above,  south/below, east/above, west/above,
     north west/above left,    north east/above right,
     south west/below left,    south east/below right}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

Shape chamfered rectangle

This shape is a rectangle with optionally chamfered corners.

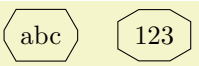



```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \node[chamfered rectangle, white, fill=red, double=red, draw, very thick]
    {\bf STOP!};
\end{tikzpicture}
```

There are PGF keys to specify how this shape is drawn (to use these keys in Ti*k*Z simply remove the `/pgf/` path).

`/pgf/chamfered rectangle angle=`⟨*angle*⟩ (no default, initially `45`)

Sets the angle *from the vertical* for the chamfer.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \tikzset{every node/.style={chamfered rectangle, draw}}
  \node[chamfered rectangle angle=30] {abc};
  \node[chamfered rectangle angle=60] at (1.5,0) {123};
\end{tikzpicture}
```

`/pgf/chamfered rectangle xsep=`⟨*length*⟩ (no default, initially `.666ex`)

Sets the distance that the chamfer extends horizontally beyond the node contents (which includes the `inner sep`). If ⟨*length*⟩ is large, such that the top and bottom chamfered edges would cross, then ⟨*length*⟩ is ignored and the chamfered edges are drawn so that they meet in the middle.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \tikzset{every node/.style={chamfered rectangle, draw}}
  \node[chamfered rectangle xsep=2pt] {def};
  \node[chamfered rectangle xsep=2cm] at (1.5,0) {456};
\end{tikzpicture}
```

`/pgf/chamfered rectangle ysep=`⟨*length*⟩ (no default, initially `.666ex`)

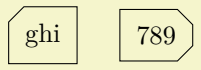
Sets the distance that the chamfer extends vertically beyond the node contents. If ⟨*length*⟩ is large, such that the left and right chamfered edges would cross, then ⟨*length*⟩ is ignored and the chamfered edges are drawn so that they meet in the middle.

`/pgf/chamfered rectangle sep=`⟨*length*⟩ (no default, initially `.666ex`)

Sets both the `xsep` and `ysep` simultaneously.

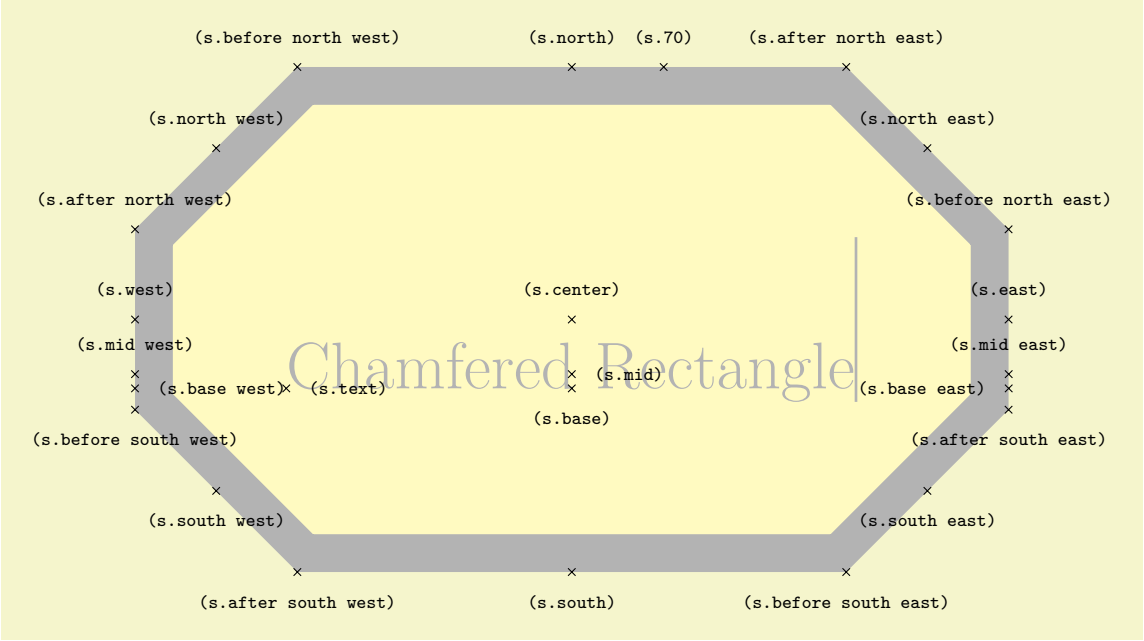
`/pgf/chamfered rectangle corners=`⟨*list*⟩ (no default, initially `chamfer all`)

Specifies which corners are chamfered. The corners are identified by their "compass point" directions (i.e. `north east`, `north west`, `south west`, and `south east`), and must be separated by commas (so if there is more than one corner in the list, it must be surrounded by braces). Any corners not mentioned in ⟨*list*⟩ are automatically not chamfered. Two additional values `chamfer all` and `chamfer none`, are also permitted.




```
\usetikzlibrary {shapes.misc}
\begin{tikzpicture}
  \tikzset{every node/.style={chamfered rectangle, draw}}
  \node[chamfered rectangle corners=north west] {ghi};
  \node[chamfered rectangle corners={north east, south east}] at (1.5,0) {789};
\end{tikzpicture}
```

The anchors for this shape are shown below (anchor `60` is an example of a border angle.




```
\usetikzlibrary {shapes.misc}
\Huge
\begin{tikzpicture}
  \node[name=s,shape=chamfered rectangle, chamfered rectangle sep=1cm,
        shape example, inner ysep=1cm, inner xsep=.75cm]
    {Chamfered Rectangle\vrule width1pt height2cm};
  \foreach \anchor/\placement in
    {text/right, center/above,    70/above,
     base/below, base east/left, base west/right,
     mid/right,  mid east/above,   mid west/above,
     north/above, south/below, east/above, west/above,
     before north east/above, north east/above, after north east/above,
     before north west/above, north west/above, after north west/above,
     before south west/below, south west/below, after south west/below,
     before south east/below, south east/below, after south east/below}
     \draw[shift=(s.\anchor)] plot[mark=x] coordinates{(0,0)}
       node[\placement] {\scriptsize\texttt{(s.\anchor)}};
\end{tikzpicture}
```

72 Spy Library: Magnifying Parts of Pictures

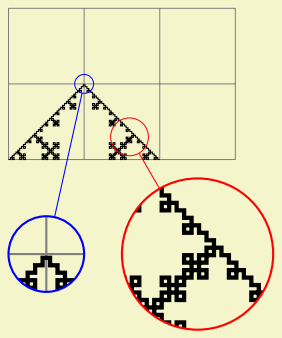
TikZ Library spy

```
\usetikzlibrary{spy} % LaTeX and plain TeX
\usetikzlibrary[spy] % ConTeXt
```

The package defines options for creating pictures in which some part of the picture is repeated in another area in a magnified way (as if you were looking through a spyglass, hence the name).

72.1 Magnifying a Part of a Picture

The idea behind the `spy` library is to make it easy to create high-density pictures in which some important parts are repeated somewhere, but magnified as if you were looking through a spyglass:



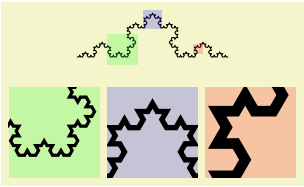
```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={circle, magnification=4, size=2cm, connect spies}]

  \draw [help lines] (0,0) grid (3,2);

  \draw [decoration=Koch curve type 1]
    decorate { decorate{ decorate{ decorate{ (0,0) -- (2,0) }}}};

  \spy [red] on (1.6,0.3)
             in node [left] at (3.5,-1.25);

  \spy [blue, size=1cm] on (1,1)
               in node [right] at (0,-1.25);
\end{tikzpicture}
```



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}[spy using overlays={size=12mm}]
  \draw [decoration=Koch snowflake]
    decorate { decorate{ decorate{ decorate{ (0,0) -- (2,0) }}}};

  \spy [green,magnification=3] on (0.6,0.1) in node at (-0.3,-1);
  \spy [blue,magnification=5]  on (1,0.5)   in node at (1,-1);
  \spy [red,magnification=10]  on (1.6,0.1) in node at (2.3,-1);
\end{tikzpicture}
```

Note that this magnification uses what is called a *canvas transformation* in this manual: Everything is magnified, including line width and text.

In order for "spying" to work, the picture obviously has to be drawn several times: Once at its normal size and then again for each "magnifying glass". Several keys and commands work in concert to make this possible:

- You need to make TikZ aware of the fact that a picture (or just a scope) is to be magnified. This is done by adding the special key `spy scope` to a `{scope}` or `{tikzpicture}` (which is also just a scope). Some special keys like `spy using outlines` implicitly set the `spy scope`.
- Inside this scope you may then use the command `\spy`, which is only available inside such scopes (so there is no danger of you inadvertently using this command outside such a scope). This command has a special syntax and will (at some point) create two nodes: One node that shows the magnified picture (called the *spy-in node*) and another node showing which part of the original picture is magnified (called the *spy-on* node). The spy-in node is, indeed, a normal node, so it can have any shape or border that you like and you can apply all of TikZ's advanced features to it. The only difference compared to a normal node is that instead of some "text" it contains a magnified version of the picture, clipped to the size of the node.

  The `\spy` command does not create the nodes immediately. Rather, the creation of these nodes is postponed till the end of the `spy scope` in which the `\spy` command is used. This is necessary since in order to repeat the whole scope inside the node containing the magnified version, the whole picture needs to be available when this node is created.

A basic question any library for "magnifying things" has to address is how you specify which part of the picture is to be magnified (the spy-on node) and where this magnified part is to be shown (the spy-in node). There are two possible ways:

1. You specify the size and position of the spy-on node. Then the size of the spy-in node is determined by the size of the spy-on node and the magnification factor – you can still decide where the spy-in node should be placed, but not its size.

2. Alternatively, you specify the size and position of the spy-in node. Then, similarly to the first case, the size of the spy-on node is determined implicitly and you can only decide where the spy-on node should be placed, but not its size.

The `spy` library uses the second method: You specify the size and position of the spy-in nodes, the sizes of the spy-on nodes are then computed automatically.

72.2 Spy Scopes

`/tikz/spy scope=`⟨*options*⟩ (default empty)

This option may be used with a `{scope}` or any environment that creates such a scope internally (like `{tikzpicture}`). It has the following effects:

- It resets a number of graphic state parameters, including the color, line style, and others. This is necessary for technical reasons.
- It tells Ti*k*Z that the content of the scope should be saved internally in a special box.
- It defines the command `\spy` so that it can be used inside the scope.
- At the end of the scope, the nodes belonging to the `\spy` commands used inside the scope are created.
- The ⟨*options*⟩ are saved in an internal style. Each time `\spy` is used, these ⟨*options*⟩ will be used.
- Three keys are defined that provide useful shortcuts:

  `/tikz/size=`⟨*dimension*⟩ (no default)

  Inside a `spy scope`, this is a shortcut for `minimum size`.

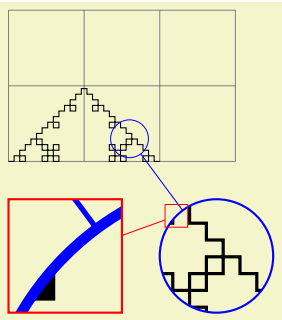
  `/tikz/height=`⟨*dimension*⟩ (no default)

  Inside a `spy scope`, this is a shortcut for `minimum height`.

  `/tikz/width=`⟨*dimension*⟩ (no default)

  Inside a `spy scope`, this is a shortcut for `minimum width`.

It is permissible to nest `spy scopes`. In this case, all `\spy` commands inside the inner `spy scope` only have an effect on material inside the scope, whereas `\spy` commands outside the inner `spy scope` but inside the outer `spy scope` allow you to "spy on the spy".



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={rectangle, red, magnification=5,
                       size=1.5cm, connect spies}]

  \begin{scope}
    [spy using outlines={circle, blue,
                         magnification=3, size=1.5cm, connect spies}]
    \draw [help lines] (0,0) grid (3,2);

    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

    \spy on (1.6,0.3) in node (zoom) [left] at (3.5,-1.25);
  \end{scope}

  \spy on (zoom.north west) in node [right] at (0,-1.25);
\end{tikzpicture}
```

72.3 The Spy Command

`\spy[`⟨*options*⟩`] on` ⟨*coordinate*⟩ `in node` ⟨*node options*⟩`;`

This command can only be used inside a `spy scope`. Let us start with the syntax:

- The `\spy` command is not a special case of `\path`. Rather, it has a small parser of its own.
- Following the optional ⟨*options*⟩, you must write `on`, followed by a coordinate. This coordinate will be the center of the area that is to be magnified.
- Following the ⟨*coordinate*⟩, you must write `in node` followed by some ⟨*node options*⟩. The syntax for these options is the same as for a normal `node` path command, such as `[left]` or `(foo) [red] at (bar)`. *However*, ⟨*node options*⟩ are *not* followed by a curly brace. Rather, the ⟨*node options*⟩ must directly be followed by a semicolon.

The effect of this command is the following: The ⟨*options*⟩, ⟨*coordinate*⟩, and ⟨*node options*⟩ are stored internally till the end of the current `spy scope`. This means that, in particular, you can reference any node inside the `spy scope`, even if it is not yet defined when the `\spy` command is given. At the end of the current `spy scope`, two nodes are created, called the *spy-in node* and the *spy-on node*.

- The *spy-in node* is the node that contains a magnified part of the picture (the node *in* which we see on what we spy). This node is, indeed, a normal TikZ node, so you can use all standard options to style this node. In particular, you can specify a shape or a border color or a drop shadow or whatever. The only thing that is special about this node is that instead of containing some normal text, its "text" is the magnified picture.

  To be precise, the picture of the `spy scope` is scaled by a certain factor, specified by the `lens` or `magnification` options discussed below, and is shifted in such a way that the ⟨*coordinate*⟩ lies at the center of the spy-on node.
- The *spy-on node* is a node that is centered on the ⟨*coordinate*⟩ and whose size reflects exactly the area shown inside the spy-in node (the node containing *on* what we spy).

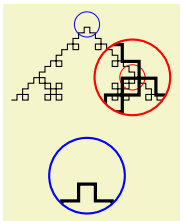
Let us now go over what happens in detail when the two nodes are created:

1. A scope is started. Two sets of options are used with this scope: First, the options passed to the enclosing `spy scope` and then the ⟨*options*⟩ (which will, thus, overrule the options of the `spy scope`).
2. Then, the spy-on node is created. However, we will first discuss the spy-in node.
3. The spy-in node is created after the spy-on node (and, hence, will cover the spy-on node in case they overlap). When this node is created, the ⟨*node options*⟩ are used in addition to the effect caused by the ⟨*options*⟩ and the options of the `{spy scope}`. Additionally, the following style is used:

   `/tikz/every spy in node` (style, no value)

   This style is used with every spy-in node.

   The position of the node (the `at` option) is set to the ⟨*coordinate*⟩ by default, so that it will cover the to-be-magnified area. You can change this by providing the `at` option yourself:



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={circle, magnification=3, size=1cm}]

  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [red]  on (1.6,0.3) in node;
  \spy [blue] on (1,1)     in node at (1,-1);
\end{tikzpicture}
```

   No "text" can be specified for the node. Rather, the "text" shown inside this node is the picture of the current `spy scope`, but canvas-transformed according to the following key:

   `/tikz/lens=`⟨*options*⟩ (no default)

   The ⟨*options*⟩ should contain transformation commands like `scale` or `rotate`. These transformations are applied to the picture when it is shown inside the spy-on node.

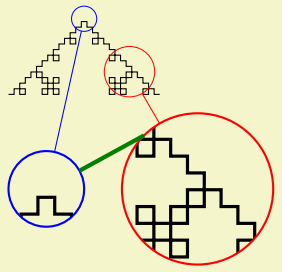
   Since the most common transformation is undoubtedly a simple scaling, there is a special style for this:

   `/tikz/magnification=`⟨*number*⟩ (no default)

   This has the same effect as saying `lens={scale=`⟨*number*⟩`}`.

Now, usually the size of a node is determined in such a way that it "fits" around the text of the node. For a spy-on node this is not a good approach since the "text" of this node would contain "the whole picture". Because of this, TikZ acts as if the "text" of the node has zero size. You must then use keys like `minimum size` to cause the node to have a certain size. Note that the key `size` is an abbreviation for `minimum size` inside a spy scope.

You can name the spy-on node in the usual ways. Additionally, the node is (also) always named `tikzspyinnode`. Following the spy scope, you can use this node like any other node:



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  \begin{scope}
    [spy using outlines={circle, magnification=3, size=2cm, connect spies}]

    \draw [decoration=Koch curve type 1]
      decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

    \spy [red] on (1.6,0.3) in node (a) [left] at (3.5,-1.25);

    \spy [blue, size=1cm] on (1,1) in node (b) [right] at (0,-1.25);
  \end{scope}
  \draw [ultra thick, green!50!black] (b) -- (a.north west);
\end{tikzpicture}
```

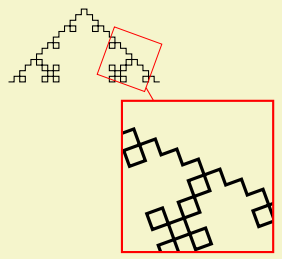
4. Once both nodes have been created, the current value of the following key is used to connect them:

   `/tikz/spy connection path`=⟨*code*⟩ (no default, initially empty)

   The ⟨*code*⟩ is executed after the spy-on and spy-in nodes have just been created. Inside this ⟨*code*⟩, the two nodes can be accessed as `tikzspyinnode` and `tikzspyonnode`. For example, the key `connect spies` sets this command to

   ```
   \draw[thin] (tikzspyonnode) -- (tikzspyinnode);
   ```

Returning to the creation of the spy-in node: This node is centered on ⟨*coordinate*⟩ (more precisely, its anchor is set to `center` and the `at` option is set to ⟨*coordinate*⟩). Its size and shape are initially determined in the same way as the size and shape of the spy-on node (unless, of course, you explicitly provide a different shape for, say, the spy-on node locally, which is not really a good idea). Then, additionally, the *inverted* transformation done by the `lens` option is applied, resulting in a node whose size and shape exactly corresponds to the area in the picture that is shown in the spy-on node.



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={lens={scale=3,rotate=20}, size=2cm, connect spies}]

  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [red] on (1.6,0.3) in node at (2.5,-1.25);
\end{tikzpicture}
```

Like for the spy-in node, a style can be used to format the spy-on node:

`/tikz/every spy on node` (style, no value)

This style is used with every spy-on node.

The spy-on node is named `tikzspyonnode` (but, as always, this node is only available after the spy scope). If you have multiple spy-on nodes and you would like to access all of them, you need to use the `name` key inside the `every spy on node` style.

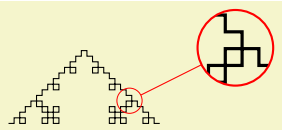
The `inner sep` and `outer sep` of both spy-in and spy-on nodes are set to `0pt`.

72.4 Predefined Spy Styles

There are some predefined styles that make using the `spy` library easier. The following two styles can be used instead of `spy scope`, they pass their ⟨*options*⟩ directly to `spy scope`. They additionally set up the graphic styles to be used for the spy-in nodes and the spy-on nodes in some special way.

/tikz/spy using outlines=⟨*options*⟩ (default empty)

This key creates a `spy scope` in which the spy-in node is drawn, but not filled, using a thick line; and the spy-on node is drawn, but not filled, using a very thin line.



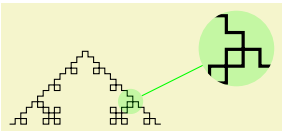
```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using outlines={circle, magnification=3, size=1cm, connect spies}]

  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [red] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

/tikz/spy using overlays=⟨*options*⟩ (default empty)

This key creates a `spy scope` in which both the spy-in and spy-on nodes are filled, but with the fill opacity set to 20%.



```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm, connect spies}]

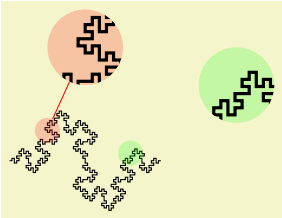
  \draw [decoration=Koch curve type 1]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [green] on (1.6,0.3) in node at (3,1);
\end{tikzpicture}
```

The following style is useful for connecting the spy-in and the spy-on nodes:

/tikz/connect spies (no value)

Causes the spy-in and the spy-on nodes to be connected by a thin line.



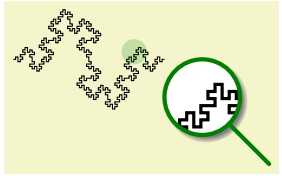
```
\usetikzlibrary {decorations.fractals,spy}
\begin{tikzpicture}
  [spy using overlays={circle, magnification=3, size=1cm}]

  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [green] on (1.6,0.1) in node at (3,1);
  \spy [red,connect spies] on (0.5,0.4) in node at (1,1.5);
\end{tikzpicture}
```

72.5 Examples

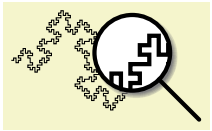
Usually, the spy-in node and the spy-on node should have the same shape. However, you might also wish to use the `circle` shape for the spy-on node and the `magnifying glass` shape for the spy-in node:



```
\usetikzlibrary {decorations.fractals,shadows,shapes.symbols,spy}
\tikzset{spy using mag glass/.style={
    spy scope={
      every spy on node/.style={
        circle,
        fill, fill opacity=0.2, text opacity=1},
      every spy in node/.style={
        magnifying glass, circular drop shadow,
        fill=white, draw, ultra thick, cap=round},
      #1
    }}}
\begin{tikzpicture}[spy using mag glass={magnification=3, size=1cm}]
  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy [green!50!black] on (1.6,0.1) in node at (2.5,-0.5);
\end{tikzpicture}
```

With the magnifying glass, you can also put it “on top” of the picture itself:



```
\usetikzlibrary {decorations.fractals,shadows,shapes.symbols,spy}
\begin{tikzpicture}
  [spy scope={magnification=4, size=1cm},
   every spy in node/.style={
     magnifying glass, circular drop shadow,
     fill=white, draw, ultra thick, cap=round}]

  \draw [decoration=Koch curve type 2]
    decorate{ decorate{ decorate{ (0,0) -- (2,0) }}};

  \spy on (1.6,0.1) in node;
\end{tikzpicture}
```

73 SVG-Path Library

Ti*k*Z Library svg.path

```
\usepgflibrary{svg.path}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[svg.path]  % ConTeXt when using pure PGF
\usetikzlibrary{svg.path} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[svg.path] % ConTeXt when using TikZ
```

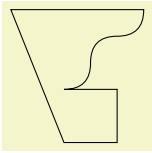
This library defines a command that allows you to specify a path using the SVG-syntax.

\pgfpathsvg{⟨*path*⟩}

This command extends the current path by a ⟨*path*⟩ given in the SVG-path-data syntax. This syntax is described in detail in Section 8.3 of the SVG-specification, Version 1.1.

In principle, the complete syntax is supported and the library just provides a parser and a mapping to basic layer commands. For instance, `M 0 10` is mapped to `\pgfpathmoveto{\pgfpoint{0pt}{10pt}}`. There are, however, a few things to be aware of:

- The computation underlying the arc commands `A` and `a` are not numerically stable, which may result in quite imprecise arcs. Bézier curves, both quadratic and cubic, are not affected, neither are arcs spanning degrees that are multiples of 90°.
- The dimensionless units of SVG are always interpreted as points (`pt`). This is a problem with paths like `M 20000 0`, which will raise an error message since TeX cannot handle dimensions larger than about 16 000 points.
- All coordinate and canvas transformations apply to the path in the usual fashion.
- The `\pgfpathsvg` command can be freely intermixed with other path commands.



```
\usepgflibrary {svg.path}
\begin{pgfpicture}
  \pgfpathsvg{M 0 0 l 20 0 0 20 -20 0 q 10 0 10 10
              t 10 10 10 10 h -50 z}
  \pgfusepath{stroke}
\end{pgfpicture}
```

74 To Path Library

Ti*k*Z Library topaths

```
\usetikzlibrary{topaths} % LaTeX and plain TeX
\usetikzlibrary[topaths] % ConTeXt
```

This library provides predefined to paths for use with the `to` path operation. After loading this package, you can say for instance `to [loop]` to add a loop to a node.

This library is loaded automatically by Ti*k*Z, so you do not need to load it yourself.

74.1 Straight Lines

The following style installs a to path that is simply a straight line from the start coordinate to the target coordinate.

/tikz/line to (no value)

Causes a straight line to be added to the path upon a `to` or an `edge` operation.


```
\tikz {\draw (0,0) to[line to] (1,0);}
```

74.2 Move-Tos

The following style installs a to path that simply "jumps" to the target coordinate.

/tikz/move to (no value)

Causes a move to be added to the path upon a `to` or an `edge` operation.


```
\tikz \draw (0,0) to[line to] (1,0)
                  to[move to] (2,0) to[line to] (3,0);
```

74.3 Curves

The `curve to` style causes the to path to be set to a curve. The exact way this curve looks can be influenced via a number of options.

/tikz/curve to (no value)

Specifies that the `to path` should be a curve. This curve will leave the start coordinate at a certain angle, which can be specified using the `out` option. It reaches the target coordinate also at a certain angle, which is specified using the `in` option. The control points of the curve are at a certain distance that is computed in different ways, depending on which options are set.

All of the following options implicitly cause the `curve to` style to be installed.

/tikz/out=⟨*angle*⟩ (no default)

The angle at which the curve leaves the start coordinate. If the start coordinate is a node, the start coordinate is the point on the border of the node at the given ⟨*angle*⟩. The control point will, thus, lie at a certain distance in the direction ⟨*angle*⟩ from the start coordinate.



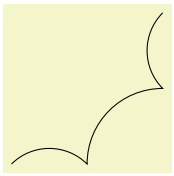
```
\begin{tikzpicture}[out=45,in=135]
  \draw (0,0) to (1,0)
        (0,0) to (2,0)
        (0,0) to (3,0);
\end{tikzpicture}
```

/tikz/in=⟨*angle*⟩ (no default)

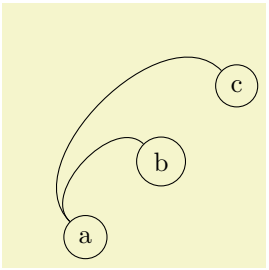
The angle at which the curve reaches the target coordinate.

/tikz/relative=⟨*true or false*⟩ (default `true`)

This option tells Ti*k*Z whether the `in` and `out` angles should be considered absolute or relative. Absolute means that an `out` angle of 30° means that the curve leaves the start coordinate at an angle of 30° relative to the paper (unless, of course, further transformations have been installed). A *relative* angle is, by comparison, measured relative to a straight line from the start coordinate to the target coordinate. Thus, a relative angle of 30° means that the curve will bend to the left from the line going straight from the start to the target. For the target, the relative coordinate is measured in the same manner, namely relative to the line going from the start to the target. Thus, an angle of 150° means that the curve will reach target coming slightly from the left.



```
\begin{tikzpicture}[out=45,in=135,relative]
  \draw (0,0) to (1,0)
              to (2,1)
              to (2,2);
\end{tikzpicture}
```

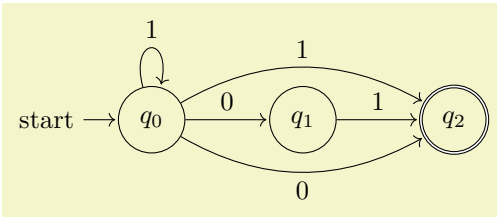



```
\begin{tikzpicture}[out=90,in=90,relative]
  \node [circle,draw] (a) at (0,0) {a};
  \node [circle,draw] (b) at (1,1) {b};
  \node [circle,draw] (c) at (2,2) {c};

  \path (a) edge (b)
            edge (c);
\end{tikzpicture}
```

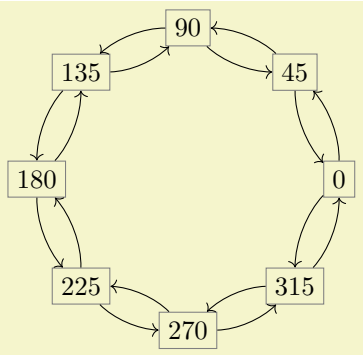
/tikz/bend left=⟨*angle*⟩ (default last value)

This option sets `out=`⟨*angle*⟩`,in=`180 − ⟨*angle*⟩`,relative`. If no ⟨*angle*⟩ is given, the last given `bend left` or `bend right` angle is used.




```
\usetikzlibrary {automata,positioning}
\begin{tikzpicture}[shorten >=1pt,node distance=2cm,on grid]
  \node[state,initial]   (q_0)                {$q_0$};
  \node[state]           (q_1) [right=of q_0] {$q_1$};
  \node[state,accepting](q_2) [right=of q_1] {$q_2$};

  \path[->] (q_0) edge              node [above]  {0} (q_1)
                  edge [loop above] node          {1} ()
                  edge [bend left]  node [above]  {1} (q_2)
                  edge [bend right] node [below]  {0} (q_2)
            (q_1) edge              node [above]  {1} (q_2);
\end{tikzpicture}
```




```
\begin{tikzpicture}
  \foreach \angle in {0,45,...,315}
    \node[rectangle,draw=black!50] (\angle) at (\angle:2) {\angle};

  \foreach \from/\to in {0/45,45/90,90/135,135/180,
                         180/225,225/270,270/315,315/0}
    \path (\from) edge [->,bend right=22,looseness=0.8] (\to)
                  edge [<-,bend left=22,looseness=0.8] (\to);
\end{tikzpicture}
```

/tikz/bend right=⟨*angle*⟩ (default last value)

Works like the `bend left` option, only the bend is to the other side.

/tikz/bend angle=⟨*angle*⟩ (no default)

Sets the angle to be used by the `bend left` or `bend right`, but without actually selecting the `curve to` or the `relative` option. This is useful for globally specifying a `bend angle` for a whole picture.

/tikz/looseness=⟨*number*⟩ (no default, initially `1`)

This number specifies how "loose" the curve will be. In detail, the following happens: Ti*k*Z computes the distance between the start and the target coordinate (if the start and/or target coordinate are nodes, the distance is computed between the points on their border). This distance is then multiplied by a fixed factor and also by the factor ⟨*number*⟩. The resulting distance, let us call it d, is then used as the distance of the control points from the start and target coordinates.

The fixed factor has been chosen in such a way that if ⟨*number*⟩ is `1`, if the `in` and `out` angles differ by 90°, then a quarter circle results:



```
\tikz \draw (0,0) to [out=0,in=-90]               (1,1);
\tikz \draw (0,0) to [out=0,in=-90,looseness=0.5] (1,1);
```

/tikz/out looseness=⟨*number*⟩ (no default)

Specifies the looseness factor for the out distance only.

/tikz/in looseness=⟨*number*⟩ (no default)

Specifies the looseness factor for the in distance only.

/tikz/min distance=⟨*distance*⟩ (no default)

If the computed distance for the start and target coordinates are below ⟨*distance*⟩, then ⟨*distance*⟩ is used instead.

/tikz/max distance=⟨*distance*⟩ (no default)

If the computed distance for the start and target coordinates are above ⟨*distance*⟩, then ⟨*distance*⟩ is used instead.

/tikz/out min distance=⟨*distance*⟩ (no default)

The minimum distance set only for the start coordinate.

/tikz/out max distance=⟨*distance*⟩ (no default)

The maximum distance set only for the start coordinate.

/tikz/in min distance=⟨*distance*⟩ (no default)

The minimum distance set only for the target coordinate.

/tikz/in max distance=⟨*distance*⟩ (no default)

The maximum distance set only for the target coordinate.

/tikz/distance=⟨*distance*⟩ (no default)

Set the minimum and maximum distance to the same value ⟨*distance*⟩. Note that this causes any computed distance d to be ignored and ⟨*distance*⟩ to be used instead.



```
\begin{tikzpicture}[out=45,in=135,distance=1cm]
  \draw (0,0) to (1,0)
        (0,0) to (2,0)
        (0,0) to (3,0);
\end{tikzpicture}
```

/tikz/out distance=⟨*distance*⟩ (no default)

Sets the minimum and maximum out distance.

/tikz/in distance=⟨*distance*⟩ (no default)

Sets the minimum and maximum in distance.

/tikz/out control=⟨*coordinate*⟩ (no default)

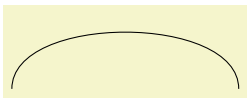
This option causes the ⟨*coordinate*⟩ to be used as the start control point. All computations of d are ignored. You can use a coordinate like `+(1,0)` to specify a point relative to the start coordinate.

/tikz/in control=⟨*coordinate*⟩ (no default)

This option causes the ⟨*coordinate*⟩ to be used as the target control point. You can use a coordinate like `+(1,0)` to specify a point relative to the *end* coordinate.

/tikz/controls=⟨*coordinate*⟩ `and` ⟨*coordinate*⟩ (no default)

This option causes the ⟨*coordinate*⟩s to be used as control points.



```
\tikz \draw (0,0) to [controls=+(90:1) and +(90:1)] (3,0);
```

74.4 Loops

/tikz/loop (no value)

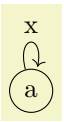
This key is similar to the `curve to` key, but differs in the following ways: First, the actual target coordinate is ignored and the start coordinate is used as the target coordinate. Thus, it is allowed not to provide any target coordinate, which can be useful with unnamed nodes. Second, the `looseness` is set to `8` and the `min distance` to `5mm`. These settings result in rather nice loops when the opening angle (difference between `in` and `out`) is 30°.




```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [in=30,out=60,loop] ();
\end{tikzpicture}
```

/tikz/loop above (style, no value)

Sets the `loop` style and sets in and out angles such that loop is above the node. Furthermore, the `above` option is set, which causes a node label to be placed at the correct position.




```
\begin{tikzpicture}
  \node [circle,draw] {a} edge [loop above] node {x} ();
\end{tikzpicture}
```

/tikz/loop below (style, no value)

Works like the previous option.

/tikz/loop left (style, no value)

Works like the previous option.

/tikz/loop right (style, no value)

Works like the previous option.

/tikz/every loop (style, initially `->,shorten >=1pt`)

This style is installed at the beginning of every loop.



```
\begin{tikzpicture}[every loop/.style={}]
  \draw (0,0) to [loop above] () to [loop right] ()
              to [loop below] () to [loop left]  ();
\end{tikzpicture}
```

75 Through Library

TikZ Library through

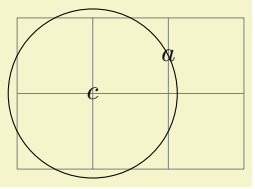
```
\usetikzlibrary{through} % LaTeX and plain TeX
\usetikzlibrary[through] % ConTeXt
```

This library defines keys for creating shapes that go through given points.

`/tikz/circle through`=⟨*coordinate*⟩ (no default)

When this key is given as an option to a node, the following happens:

1. The `inner sep` and the `outer sep` are set to zero.
2. The shape is set to `circle`.
3. The `minimum size` is set such that the circle around the center of the node (which is specified using `at`), goes through ⟨*coordinate*⟩.




```
\usetikzlibrary {through}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \node (a) at (2,1.5) {$a$};
  \node [draw] at (1,1) [circle through={(a)}] {$c$};
\end{tikzpicture}
```

76 Tree Library

TikZ Library trees

```
\usetikzlibrary{trees} % LaTeX and plain TeX
\usetikzlibrary[trees] % ConTeXt
```

This packages defines styles to be used when drawing trees.

76.1 Growth Functions

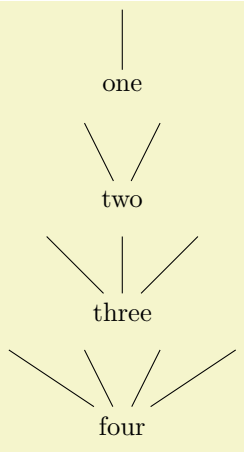
The package `trees` defines two new growth functions. They are installed using the following options:

`/tikz/grow via three points=one child at (⟨x⟩ ) and two children at (⟨y⟩) and (⟨z⟩)` (no default)

This option installs a growth function that works as follows: If a parent node has just one child, this child is placed at $\langle x \rangle$. If the parent node has two children, these are placed at $\langle y \rangle$ and $\langle z \rangle$. If the parent node has more than two children, the children are placed at points that are linearly extrapolated from the three points $\langle x \rangle$, $\langle y \rangle$, and $\langle z \rangle$. In detail, the position is $x + \frac{n-1}{2}(y - x) + (c - 1)(z - y)$, where n is the number of children and c is the number of the current child (starting with 1).

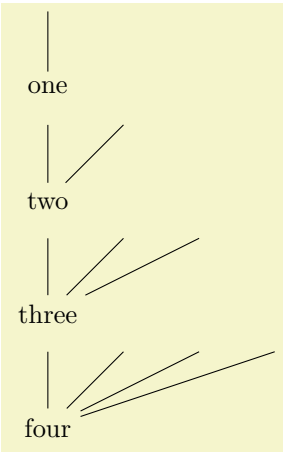
The net effect of all this is that if you have a certain "linear arrangement" in mind and use this option to specify the placement of a single child and of two children, then any number of children will be placed correctly.

Here are some arrangements based on this growth function. We start with a simple "above" arrangement:



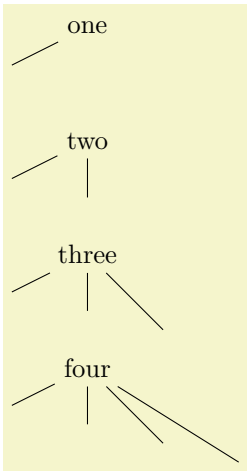

```
\usetikzlibrary {trees}
\begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (-.5,1) and (.5,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

The next arrangement places children above, but "grows only to the right".




```
\usetikzlibrary {trees}
\begin{tikzpicture}[grow via three points={%
    one child at (0,1) and two children at (0,1) and (1,1)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

In the final arrangement, the children are placed along a line going down and right.




```
\usetikzlibrary {trees}
\begin{tikzpicture}[grow via three points={%
    one child at (-1,-.5) and two children at (-1,-.5) and (0,-.75)}]
  \node at (0,0) {one} child;
  \node at (0,-1.5) {two} child child;
  \node at (0,-3) {three} child child child;
  \node at (0,-4.5) {four} child child child child;
\end{tikzpicture}
```

These examples should make it clear how you can create new styles to arrange your children along a line.

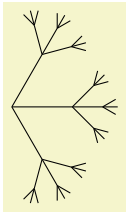
`/tikz/grow cyclic` (no value)

This style causes the children to be arranged "on a circle". For this, the children are placed at distance `\tikzleveldistance` from the parent node, but not on a straight line, but on points on a circle. Instead of a sibling distance, there is a `sibling angle` that denotes the angle between two given children.

`/tikz/sibling angle=`⟨*angle*⟩ (no default)

Sets the angle between siblings in the `grow cyclic` style.

Note that this function will rotate the coordinate system of the children to ensure that the grandchildren will grow in the right direction.

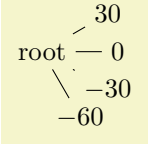


```
\usetikzlibrary {trees}
\begin{tikzpicture}
  [grow cyclic,
   level 1/.style={level distance=8mm,sibling angle=60},
   level 2/.style={level distance=4mm,sibling angle=45},
   level 3/.style={level distance=2mm,sibling angle=30}]
  \coordinate [rotate=-90] % going down
    child foreach \x in {1,2,3}
      {child foreach \x in {1,2,3}
        {child foreach \x in {1,2,3}}};
\end{tikzpicture}
```

`/tikz/clockwise from=`⟨*angle*⟩ (no default)

This option also causes children to be arranged on a circle. However, the rule for placing children is simpler than with the `grow cyclic` style: The first child is placed at ⟨*angle*⟩ at a distance of `\tikzleveldistance`. The second child is placed at the same distance from the parent, but at angle ⟨*angle*⟩ − `\tikzsiblingangle`. The third child is displaced by another `\tikzsiblingangle` in a clockwise fashion, and so on.

Note that this function will not rotate the coordinate system.




```
\usetikzlibrary {trees}
\begin{tikzpicture}
  \node {root}
  [clockwise from=30,sibling angle=30]
  child {node {$30$}}
  child {node {$0$}}
  child {node {$-30$}}
  child {node {$-60$}};
\end{tikzpicture}
```

`/tikz/counterclockwise from=`⟨*angle*⟩ (no default)

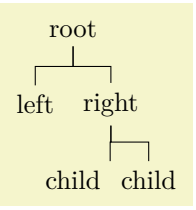
Works the same way as `clockwise from`, but sibling angles are added instead of subtracted.

76.2 Edges From Parent

The following styles can be used to modify how the edges from parents are drawn:

`/tikz/edge from parent fork down` (style, no value)

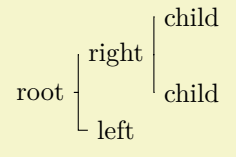
This style will draw a line from the parent downwards (for half the level distance) and then on to the child using only horizontal and vertical lines.




```
\usetikzlibrary {trees}
\begin{tikzpicture}
  \node {root}
    [edge from parent fork down]
    child {node {left}}
    child {node {right}
      child[child anchor=north east] {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

`/tikz/edge from parent fork right` (style, no value)

This style behaves similarly, only it will first draw its edge to the right.




```
\usetikzlibrary {trees}
\begin{tikzpicture}
  \node {root}
    [edge from parent fork right,grow=right]
    child {node {left}}
    child {node {right}
      child {node {child}}
      child {node {child}}
    };
\end{tikzpicture}
```

`/tikz/edge from parent fork left` (style, no value)

Behaves similarly to the previous styles.

`/tikz/edge from parent fork up` (style, no value)

Behaves similarly to the previous styles.

77 Turtle Graphics Library

TikZ Library turtle

```
\usetikzlibrary{turtle} % LaTeX and plain TeX
\usetikzlibrary[turtle] % ConTeXt
```

This little library defines some keys to create simple turtle graphics in the tradition of the Logo programming language. These commands are mostly for fun, but they can also be used for more "serious" business.



```
\usetikzlibrary {turtle}
\tikz[turtle/distance=2mm]
  \draw [turtle={home,forward,right,forward,left,forward,left,forward}];
```

Even though the `turtle` keys looks like an option, it uses the `insert path` option internally to produce a path.

The basic drawing model behind the turtle graphics is very simple: There is a (virtual) *turtle* that crawls around the page, thereby extending the path. The turtle always heads in a certain direction. When you move the turtle forward, you extend the path in that direction; turning the turtle just changes the direction, it does not cause anything to be drawn.

The turtle always moves relative to the last current point of the path and you can mix normal path commands with turtle commands. However, the direction of the turtle is managed independently of other path commands.

`/tikz/turtle=`⟨*keys*⟩ (no default)

This key executes the ⟨*keys*⟩ with the current key path set to `/tikz/turtle`.



```
\usetikzlibrary {turtle}
\tikz[turtle/distance=2mm]
  \draw [turtle={home,fd,rt,fd,lt,fd,lt,fd}];
```

`/tikz/turtle/home` (no value)

Places the turtle at the origin and lets it head upward.

`/tikz/turtle/forward=`⟨*distance*⟩ (default see text)

Makes the turtle move forward by the given ⟨*distance*⟩. If no ⟨*distance*⟩ is specified, the current value of the following key is used:

`/tikz/turtle/distance=`⟨*distance*⟩ (no default, initially `1cm`)

The default distance by which the turtle advances.

"Moving forward the turtle" actually means that, relative to the current last point on the path, a point at the given ⟨*distance*⟩ in the direction the turtle is currently heading is computed. Then, the operation `to[turtle/how]` is used to extend the path to this point.

`/tikz/turtle/how` (style, initially empty)

This style can set up the `to path` used by turtles. By setting this style you can change the to-path:



```
\usetikzlibrary {turtle}
\tikz \draw [turtle={how/.style={bend left},home,forward,right,forward}];
```

`/tikz/turtle/fd` (no value)

An abbreviation for the `forward` key.

`/tikz/turtle/back=`⟨*distance*⟩ (default see text)

This has the same effect as a `turtle/forward` for the negated ⟨*distance*⟩ value.

/tikz/turtle/bk (no value)

An abbreviation for the `back` key.

/tikz/turtle/left=⟨*angle*⟩ (default 90)

Turns the turtle left by the given angle.

/tikz/turtle/lt (no value)

An abbreviation for the `left` key.

/tikz/turtle/right=⟨*angle*⟩ (default 90)

Turns the turtle right by the given angle.

/tikz/turtle/rt (no value)

An abbreviation for the `right` key.

Turtle graphics are especially nice in conjunction with the `\foreach` statement:



```
\usetikzlibrary {turtle}
\tikz \filldraw [thick,blue,fill=blue!20]
  [turtle=home]
  \foreach \i in {1,...,5}
  {
    [turtle={forward,right=144}]
  };
```

78 Views Library

Ti*k*Z Library views

```
\usetikzlibrary{views} % LaTeX and plain TeX
\usetikzlibrary[views] % ConTeXt
```

This library is used for creating *views*, which are transformations of a part of a picture so that this part "fits" into a "viewbox". Mostly, views are useful in conjunction with animations.

A *view* is essentially a "window" through which you see a graphic. To establish a view, you specify a rectangle – which is the window – and another rectangle surrounding the to-be-viewed graphic. The graphic will then be rescaled and shifted in such a way that the to-be-viewed rectangle matches the view's rectangle as well as possible. Note that establishing a view causes a *canvas* transformation to be installed, not a coordinate transformation.

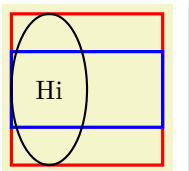
View boxes are only seldom needed in normal graphics; you may prefer to use coordinate transformations or the `spy` library. Their main application is with animations since you can *animate* the to-be-viewed rectangle. This makes it easy to create animations in which you zoom in, zoom out, and pan a graphic.

`/tikz/meet=`⟨*to-be-viewed corner*⟩ `rectangle` ⟨*to-be-viewed corner*⟩ `at` ⟨*window corner*⟩ `rectangle` ⟨*window corner*⟩ (no default)

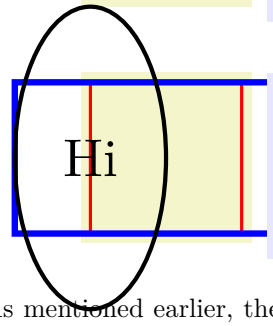
Use this key with a scope to establish a view for the scope. In the argument to the `meet` key, both `rectangle` texts are optional. Also, everything following `at` is optional; when it is missing, the ⟨*window corner*⟩s are assumed to be the same as the ⟨*to-be-viewed corner*⟩s. The latter are two corners of a rectangle that should be transformed in such a way that it fits inside the rectangle described by the two window corners.

More precisely, at the beginning of the scope a canvas transformation is installed that scales and translates the canvas is such a way that

1. the center of the to-be-viewed rectangle lies at the center of the window rectangle and
2. the to-be-viewed rectangle has maximum size that it still fits inside the window rectangle.

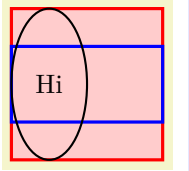



```
\usetikzlibrary {views}
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{scope}[meet = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```



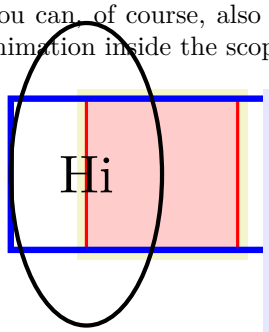

```
\usetikzlibrary {views}
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{scope}[slice = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```

As mentioned earlier, the main use of views is in conjunction with animations. In order to animate a view, you specify the scope containing the `meet` command as the target object and then animate its `:view` attribute:




```
\usetikzlibrary {animations,views}
\tikz [animate = {
  my scope:view = {
    begin on = { click, of next = here },
    0s = "{(0.5,0.5) (2.5,1.5)}",
    2s = "{(0.5,0) (1.5,2)}", forever
  }}] {
  \draw [red, fill=red!20, very thick, name=here]
    (0,0) rectangle (20mm,20mm);
  \begin{scope}[name = my scope,
                meet = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```

You can, of course, also specify the animation using the `animate myself:` key when you specify the animation inside the scope:




```
\usetikzlibrary {animations,views}
\tikz [animate = {
  my scope:view = {
  }}] {
  \draw [red, fill=red!20, very thick, name=here]
    (0,0) rectangle (20mm,20mm);
  \begin{scope}[animate = { myself: = { :view = {
                  begin on = { click, of = here },
                  0s = "{(0.5,0.5) (2.5,1.5)}",
                  2s = "{(0.5,0) (1.5,2)}", forever }}},
                slice = {(0.5,0.5) (2.5,1.5) at (0,0) (2,2)}]
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [x radius=5mm, y radius=10mm] node {Hi};
  \end{scope} }
```

`/tikz/view` (no value)

This is an alias for `/tikz/meet`.

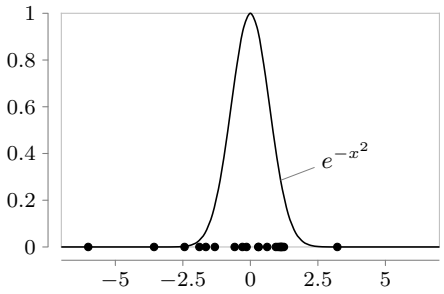
`/tikz/slice=`⟨*to-be-viewed corner*⟩ `rectangle` ⟨*to-be-viewed corner*⟩ `at` ⟨*window corner*⟩ `rectangle` ⟨*window corner*⟩ (no default)

This key works exactly like `meet`, only the second rule is changed:

2′. the to-be-viewed rectangle has minimal size that it encompasses all of the window rectangle.

Part VI
Data Visualization

by Till Tantau



- $\sum_{i=1}^{10} x_i$, where $x_i \sim U(-1, 1)$

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes=clean]
[
  visualize as smooth line=Gaussian,
  Gaussian={pin in data={text={$e^{-x^2}$},when=x is 1}}
]
data [format=function] {
  var x : interval [-7:7] samples 51;
  func y = exp(-\value x*\value x);
}
[
  visualize as scatter,
  legend={south east outside},
  scatter={
    style={mark=*,mark size=1.4pt},
    label in legend={text={
        $\sum_{i=1}^{10} x_i$, where $x_i \sim U(-1,1) $}}}
]
data [format=function] {
  var i : interval [0:1] samples 20;
  func y = 0;
  func x = (rand + rand + rand + rand + rand +
            rand + rand + rand + rand + rand);
};
```

79 Introduction to Data Visualization

Data visualization is the process of converting *data points,* which typically consist of multiple numerical values, into a graphical representation. Examples include the well-known function plots, but pie charts, bar diagrams, box plots, or vector fields are also examples of data visualizations.

The data visualization subsystem of PGF takes a general, open approach to data visualization. Like everything else in PGF, there is a powerful, but not-so-easy-to-use basic layer in the data visualization system and a less flexible, but much simpler-to-use frontend layer. The present section gives an overview of the basic ideas behind the data visualization system.

79.1 Concept: Data Points

The most important input for a data visualization is always raw data. This data is typically present in different formats and the data visualization subsystem provides methods for reading such formats and also for defining new input formats. However, independently of the input format, we may ask what kind of data the data visualization subsystem should be able to process. For two-dimensional plots we need lists of pairs of real numbers. For a bar plot we usually need a list of numbers, possibly together with some colors and labels. For a surface plot we need a matrix of triples of real numbers. For a vector field we need even more complex data.

The data visualization subsystem makes no assumption concerning which kind of data is being processed. Instead, the whole "rendering pipeline" is centered around a concept called the *data point*. Conceptually, a data point is an arbitrarily complex record that represents one piece of data that should be visualized. Data points are *not* just coordinates in the plane or the numerical values that need to be visualized. Rather, they represent the basic units of the data that needs to be visualized.

Consider the following example: In an experiment we drive a car along a road and have different measurement instruments installed. We measure the position of the car, the time, the speed, the direction the car is heading, the acceleration, and perhaps some further values. A data point would consist of a record consisting of a timestamp together with the current position of the car (presumably two or three numbers), the speed vector (another two or three numbers), the acceleration (another two or three numbers), and perhaps the label text of the current experiment.

Data points should be "information rich". They might even contain more information than what will actually be visualized. It is the job of the rendering pipeline to pick out the information relevant to one particular data visualization – another visualization of the same data might pick different aspects of the data points, thereby hopefully allowing new insights into the data.

Technically, there is no special data structure for data points. Rather, when a special macro called `\pgfdatapoint` is called, the "totality" of all currently set keys with the `/data point/` prefix in the current scope forms the data point. This is both a very general approach and quite fast since no extra data structures need to be created.

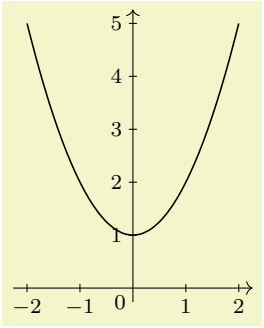
79.2 Concept: Visualization Pipeline

The *visualization pipeline* is a series of actions that are performed on the to-be-visualized data. The data is presented to the visualization pipeline in the form of a long stream of complex data points. The visualization pipeline makes several passes over this stream of data points. During the first pass(es), called the *survey phase(s)*, information is gathered about the data points such as minimal and maximal values, which can be useful for automatic fitting of the data into a given area. In the main pass over the data, called the *visualization phase*, the data points are actually visualized, for instance in the form of lines or points.

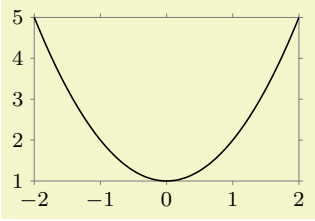
Like as for data points, the visualization pipeline makes no assumptions concerning what kind of visualization is desired. Indeed, one could even use it to produce a plain-text table. This flexibility is achieved by extensive use of objects and signals: When a data visualization starts, a number of signals (see Section 98.9 for an introduction to signals) are initialized. Then, numerous "visualization objects" are created that listen to these signals. These objects are all involved in processing the data points. For instance, the job of an `interval mapper` object is to map one attribute of a data point, such as a car's velocity, to another, such as the y-axis of a plot. For each data point the different signals are raised in a certain order and the different visualization objects now have a chance of preparing the data point for the actual visualization. Continuing the above example, there might be a second `interval mapper` that takes the computed y-position and applies a logarithm to it, because a log-plot was requested. Then another mapper, this time a `polar mapper` might be used to map everything to polar coordinates. Following this, a `plot mark visualizer` might actually draw something at the computed position.

The whole idea behind the rendering pipeline is that new kinds of data visualizations can be implemented, ideally, just by adding one or two new objects to the visualization pipeline. Furthermore, different kinds of plots can be combined in novel ways in this manner, which is usually very hard to do otherwise. For instance, the visualization pipeline makes it easy to create, say, polar-semilog-box-plots. At first sight, such new kinds of plots may seem frivolous, but data visualization is all about gaining insights into the data from as many different angles as possible.

Naturally, creating new classes and objects for the rendering pipeline is not trivial, so most users will just use the existing classes, which should, thus, be as flexible as possible. But even when one only intends to use existing classes, it is still tricky to setup the pipeline correctly since the ordering is obviously important and since things like axes and ticks need to be configured and taken care of. For this reason, the frontend libraries provide preconfigured rendering pipelines so that one can simply say that a data visualization should look like a `line plot` with `school book axes` or with `scientific axes`, which selects a certain visualization pipeline that is appropriate for this kind of plot:




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}[scale=.7]
  \datavisualization [school book axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-2:2];
    func y = \value x*\value x + 1;
  };
\end{tikzpicture}
```




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}[scale=.7]
  \datavisualization [scientific axes, visualize as smooth line]
  data [format=function] {
    var x : interval [-2:2];
    func y = \value x*\value x + 1;
  };
\end{tikzpicture}
```

One must still configure such a plot (choose styles and themes and also specify which attributes of a data point should be used), but on the whole the plot is quite simple to specify.

80 Creating Data Visualizations

80.1 Overview

The present section explains how a data visualization is created in Ti*k*Z. For this, you need to include the `datavisualization` library and then use the command `\datavisualization` whose syntax is explained in the rest of the present section. This command is part of the following library:

Ti*k*Z Library `datavisualization`

```
\usetikzlibrary{datavisualization} % LaTeX and plain TeX
\usetikzlibrary[datavisualization] % ConTeXt
```

This library must be loaded if you wish to use the `\datavisualization` command. It defines all styles needed to create basic data visualizations; additional, more specialized libraries need to be loaded for more advanced features.

In order to visualize, you basically need to do three things:

1. You need to select what kind of plot you would like to have (a "school book plot" or a "scientific 2d plot" or a "scientific spherical plot" etc.). This is done by passing an option to the `\datavisualization` command that selects this kind of plot.
2. You need to provide data points, which is done using the `data` command.
3. Additionally, you can add options that give you more fine-grained control over the way the visualization will look. You can configure the number of ticks and grid lines, where the labels are placed, the colors, or the fonts. Indeed, since the data visualization engine internally uses Ti*k*Z-styles, you can have extremely fine-grained control over how a plot will look like.

The syntax of the `\datavisualization` command is designed in such a way that you only need to provide very few options to create plots that "look good by default".

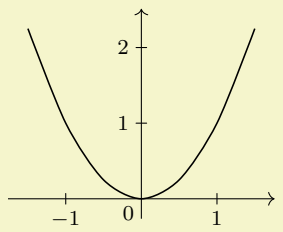
This section is structured as follows: First, the philosophy behind concepts like "data points", "axes", or "visualizers" is explained. Each of these concepts is further detailed in later section. Then, the syntax of the `\datavisualization` command is covered. The reference sections explain which predefined plot kinds are available.

80.2 Concept: Data Points and Data Formats

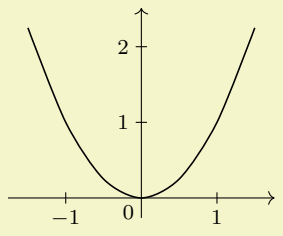
As explained in Section 79.1, data points are the basic entities that are processed by the data visualization engine. In order to specify data points, you use the `data` command, whose syntax is explained in more detail in Section 80.6. The `data` command allows you to either specify points "inline", directly inside your TeX-file; or you can specify the name of file that contains the data points.

Specifying data points. Data points can be formatted in different ways. For instance, in the so called *comma separated values* format, there is one line for each data point and the different attributes of a data point are separated by commas. Another common format is to specify data points using the so called *key–value* format, where on each line the different attributes of a data point are set using a comma-separated list of strings of the form `attribute=value`.

Here are two examples, where similar data is given in different formats:




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
    data {
      x, y
      -1.5, 2.25
      -1, 1
      -.5, .25
      0, 0
      .5, .25
      1, 1
      1.5, 2.25
    };
\end{tikzpicture}
```




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

In the first example, no format needed to be specified explicitly since the default format is the one used for the data following the `data` keyword: A list of comma-separated values, where each line represents a data point.

Number accuracy. Data visualizations typically demand a much higher accuracy and range of values than TEX provides: TEX numbers are limited to 13 bits for the integer part and 16 bits for the fractional part. Because of this, the data visualization engine does not use PGF's standard representation of numbers and TEX dimensions and does not use the standard parser when reading numbers in a data point. Instead, the `fpu` library, described in Section 56, is used to handle numbers.

This use of the `fpu` library has several effects that users of the data visualization system should be aware of:

1. You can use numbers like `100000000000000` or `0.00000000001` in data points.
2. Since the `fpu` library does not support advanced parsing, you currently *cannot* write things like `3+2` in a data point number. This will result in an error.
3. However, there is a loop-hole: If a "number" in a data point starts with a parenthesis, the value between the parentheses *is* parsed using the normal parser:
   - `100000` is allowed.
   - `2+3` yields an error.
   - `(2+3)` is allowed and evaluates to `5`.
   - `(100000)` yields an error since 100 000 is beyond the normal parser's precision.

   The bottom line is that any normal calculations should be set inside round parentheses, while large numbers should not be surrounded by parentheses. Hopefully, in the future, these restrictions will be lifted.

Section 81 gives an in-depth coverage of the available data formats and explains how new data formats can be defined.

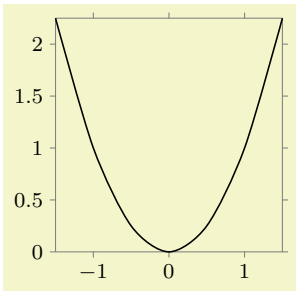
80.3 Concept: Axes, Ticks, and Grids

Most plots have two or three axes: A horizontal axis usually called the x-axis, a vertical axis called the y-axis, and possibly some axis pointing in a sloped direction called the z-axis. Axes are usually drawn as lines with *ticks* indicating interesting positions on the axes. The data visualization engine gives you detailed control over where these ticks are rendered and how many of them are used. Great care is taken to ensure that the position of ticks are chosen well by default.

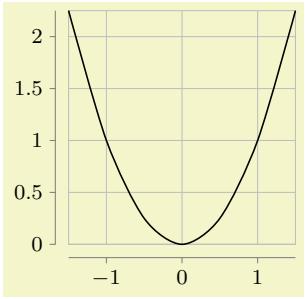
From the point of view of the data visualization engine, axes are a somewhat more general concept than "just" lines that point "along" some dimension: The data visualization engine uses axes to visualize any change of an attribute by varying the position of data points in the plane. For instance, in a polar plot, there is an "axis" for the angle and another "axis" for the distance if the point from the center. Clearly these axes vary the position of data points in the plane according to some attribute of the data points; but just as clearly they do not point in any "direction".

A great benefit of this approach is that the powerful methods for specifying and automatic inference of "good" positions for ticks or grid lines apply to all sorts of situations. For instance, you can use it to automatically put ticks and grid lines at well-chosen angles of a polar plot.

Typically, you will not need to specify axes explicitly. Rather, predefined styles take care of this for you:




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [
    scientific axes,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
  ]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

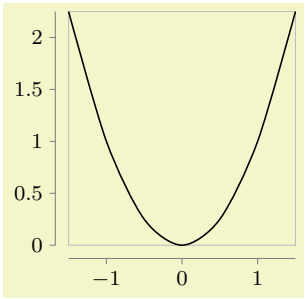



```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    all axes={grid},
    visualize as smooth line
  ]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

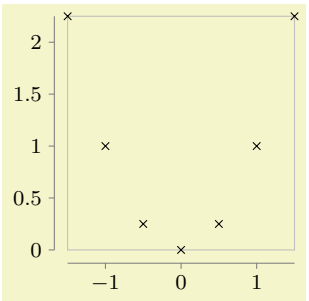
Section 82 explains in more detail how axes, ticks, and grid lines can be chosen and configured.

80.4 Concept: Visualizers

Data points and axes specify *what* is visualized and *where*. A *visualizer* specifies *how* they are visualized. One of the most common visualizers is a *line visualizer* which connects the positions of the data points in the plane using a line. Another common visualizer is the *scatter plot visualizer* where small marks are drawn at the positions of the data points. More advanced visualizers include, say, box plot visualizers or pie chart visualizers.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as smooth line
  ]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [
    scientific axes=clean,
    x axis={length=3cm, ticks=few},
    visualize as scatter
  ]
    data [format=function] {
      var x : interval [-1.5:1.5] samples 7;
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

Section 83 provides more information on visualizers as well as reference lists.

80.5 Concept: Style Sheets and Legends

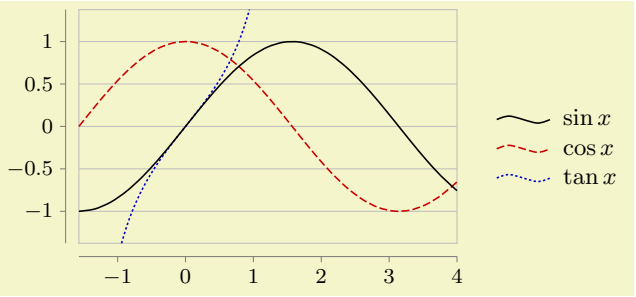
A single data visualizations may use more than one visualizer. For instance, if you wish to create a plot containing several lines, a separate visualizer is used for each line. In this case, two problems arise:

1. You may wish to make it easy for the reader to differentiate between the different visualizers. For instance, one line should be black, another should be red, and another blue. Alternatively, you might wish one line to be solid, another to be dashed, and a third to be dotted.

   Specifying such styles is trickier than one might expect; experience shows that many plots use ill-chosen and inconsistent styling. For this reason, the data visualization introduces the notion of *style sheets* for visualizers and comes with some well-designed predefined style sheets.

2. You may wish to add information concerning what the different visualizers represent. This is typically done using a legend, but it is even better to add labels directly inside the visualization. Both approaches are supported.

An example where three functions are plotted and a legend is added is shown below. Two style sheets are used so that *both* the coloring and the dashing is varied.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}[baseline]
  \datavisualization [ scientific axes=clean,
                       y axis=grid,
                       visualize as smooth line/.list={sin,cos,tan},
                       style sheet=strong colors,
                       style sheet=vary dashing,
                       sin={label in legend={text=$\sin x$}},
                       cos={label in legend={text=$\cos x$}},
                       tan={label in legend={text=$\tan x$}},
                       data/format=function ]
  data [set=sin] {
    var x : interval [-0.5*pi:4];
    func y = sin(\value x r);
  }
  data [set=cos] {
    var x : interval [-0.5*pi:4];
    func y = cos(\value x r);
  }
  data [set=tan] {
    var x : interval [-0.3*pi:.3*pi];
    func y = tan(\value x r);
  };
\end{tikzpicture}
```

Section 84 details style sheets and legends.

80.6 Usage

Inside a Ti*k*Z picture you can use the `\datavisualization` command to create a data visualization. You can use this command several times in a picture to create pictures containing multiple data visualizations.

`\datavisualization`[⟨*data visualization options*⟩]⟨*data specification*⟩;

This command is available only inside a `{tikzpicture}` environment.

The ⟨*data visualization options*⟩ are used to configure the data visualization, that is, how the data is to be depicted. The options are executed with the path prefix `/tikz/data visualization`. This means that normal Ti*k*Z options like `thin` or `red` cannot be used here. Rather, a large number of options specific to data visualizations are available.

As a minimum, you should specify at least two options: First, you should use an option that selects an axis system that is appropriate for your plot. Typical possible keys are `school book axes` or `scientific axes`, detailed information on them can be found in Section 82.

Second, you use an option to select *how* the data should be visualized. This is done using a key like `visualize as line` which will, as the name suggests, visualize the data by connecting data points in the plane using a line. Similarly, `visualize as smooth cycle` will try to fit a smooth cycle through the data points. Detailed information on possible visualizers can be found in Section 83.

Following these options, the ⟨*data specification*⟩ is used to provide the actual to-be-visualized data. The syntax is somewhat similar to commands like `\path`: The ⟨*data specification*⟩ is a sequence of keywords followed by local options and parameters, terminated with a semicolon. (Indeed, like for the `\path` command, the ⟨*data visualizers options*⟩ need not be specified at the beginning, but additional option surrounded by square brackets may be given anywhere inside the ⟨*data specification*⟩.)

The different possible keywords inside the ⟨*data specification*⟩ are explained in the following.

`\datavisualization` … `data[`⟨*options*⟩`]{`⟨*inline data*⟩`}` …;

This command is used to specify data for the data visualization. It can be used several times inside a single visualization and each time the to-be-read data may have a different format, but the data will be visualized as if it have been specified inside a single `data` command.

The behavior of the `data` command depends on whether the ⟨*inline data*⟩ is present. If it is not present, the ⟨*options*⟩ must be used to specify a source file from which the data is read; if the ⟨*inline data*⟩ is present no file will be used, instead the data should directly reside inside the TEX-file and be given between the curly braces surrounding the ⟨*inline data*⟩.

The ⟨*options*⟩ are executed with the prefix `/pgf/data`. The following options are always available:

`/pgf/data/read from file=`⟨*filename*⟩ (no default, initially empty)

If you set the `source` attribute to a non-empty ⟨*filename*⟩, the data will be read from this file. In this case, no ⟨*inline data*⟩ may be present, not even empty curly braces should be provided.

```
\datavisualization ...
  data [read from file=file1.csv]
  data [read from file=file2.csv];
```

The other way round, if `read from file` is empty, the data must directly follow as ⟨*inline data*⟩.

```
\datavisualization ...
  data {
    x, y
    1, 2
    2, 3
  };
```

The second important key is `format`, which is used to specify the data format:

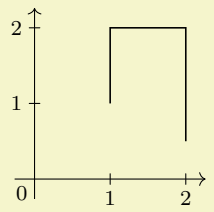
`/pgf/data/format=`⟨*format*⟩ (no default, initially `table`)

Use this key to locally set the format used for parsing the data, see Section 81 for a list of predefined formats.

The default format is the `table`-format, also known as "comma-separated values". The first line contains names of attributes separated by commas, all following lines constitute a data point where the attributes are given by the comma-separated values in that line.

Presetting attributes. Normally, the inline data or the external data contains for each data point the values of the different attributes. However, sometimes you may also wish to set an attribute to a fixed value for all data points of a data set. Suppose, for instance, that you have to source files `experiment007.csv` and `experiment023.csv` and you would like that for all data points of the first file the attribute `/data point/experiment id` is set to 7 while for the data points of the second file they are set to 23. In this case, you can specify the desired settings using an absolute path inside the ⟨*options*⟩. The effect will be local to the current `data` command:

```
\datavisualization...
  data [/data point/experiment=7,  read from file=experiment007.csv]
  data [/data point/experiment=23, read from file=experiment023.csv];
```




```
\usetikzlibrary {datavisualization}
\tikz
  \datavisualization [school book axes, visualize as line]
    data [/data point/x=1] {
      y
      1
      2
    }
    data [/data point/x=2] {
      y
      2
      0.5
    };
```

Setting options for multiple `data` commands. You may wish to generally set the format once and for all. This can be done by using the following key:

`/tikz/every data` (style, no value)

This key is executed for every `data` command.

Another way of passing options to multiple `data` commands is to use the following facility: Whenever an option with the path `/tikz/data visualization/data` is used, the path will be remapped to `/pgf/data`. This means, in particular, that you can pass an option like `data/format=table` to the `\datavisualization` command to set the data format for all `data` commands of the data visualization.

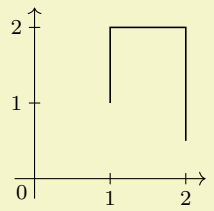
Parsing inline data. When you specify data inline, TEX needs to read the data "line-by-line", while TEX normally largely ignores end-of-line characters. For this reason, the data visualization system temporarily changes the meaning of the end-of-line character. This is only possible if TEX has not already processed the data in some other way (namely as the parameter to some macro).

The bottom line is that you cannot use inline data when the whole `\datavisualization` command is passed as a parameter to some macro that is not setup to handle "fragile" code. For instance, in a BEAMER `frame` you need to add the `fragile` option when a data visualization contains inline data.

The problem does not arise when an external data `source` is specified.

`\datavisualization` … `data point[`⟨*options*⟩`]` … `;`

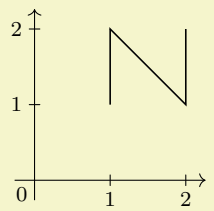
This command is used to specify data a single data point. The ⟨*options*⟩ are simply executed with the path `/data point` and then a data point is created. This means that inside the ⟨*options*⟩ you just specify the values of all attributes in key–value syntax.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [school book axes, visualize as line]
  data point [x=1, y=1]    data point [x=1, y=2]
  data point [x=2, y=2]    data point [x=2, y=0.5];
```

`/tikz/data visualization/data point=`⟨*options*⟩ (no default)

This key is the "key version" of the previous command. The difference is that this key can be used internally inside styles.




```
\usetikzlibrary {datavisualization}
\tikzdatavisualizationset{
  horizontal/.style={
    data point={x=#1, y=1}, data point={x=#1, y=2}},
}
\tikz \datavisualization
[ school book axes, visualize as line,
  horizontal=1,
  horizontal=2 ];
```

\datavisualization … data group[⟨*options*⟩]{⟨*name*⟩}+={⟨*data specifications*⟩} …;

You can store a whole ⟨*data specification*⟩ in a *data group*. This allows you to reuse data in multiple places without having to write the data to an external file.

The syntax of this command comes in the following three variants:

- `data group` [⟨*options*⟩] {⟨*name*⟩} = {⟨*data specifications*⟩}
- `data group` [⟨*options*⟩] {⟨*name*⟩} += {⟨*data specifications*⟩}
- `data group` [⟨*options*⟩] {⟨*name*⟩}

In the first case, a new data group called ⟨*name*⟩ is created (an existing data group of the same name will be erased) and the following ⟨*data specifications*⟩ is stored in this data group. The data group will not be fed to the rendering pipeline, but it is parsed at this point as if it were. The defined data group is defined globally, so you can used it in subsequent visualizations. The ⟨*options*⟩ are saved with the parsed ⟨*data specifications*⟩.

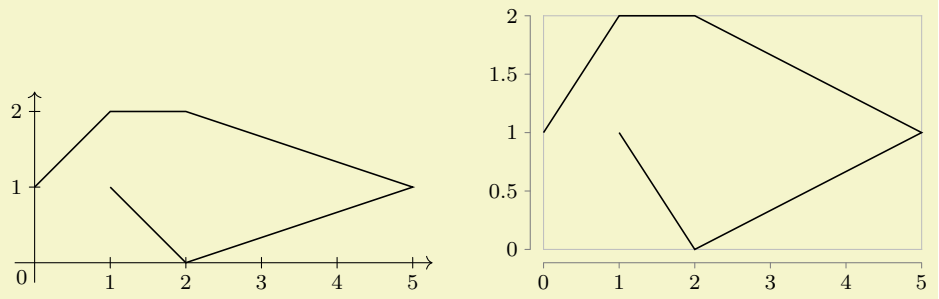
In the second case, an already existing data group is extended by adding the ⟨*data specifications*⟩ to it.

In the third case (detected by noting that the ⟨*name*⟩ is neither followed by an equal sign nor a plus sign), the contents of the previously defined data group ⟨*name*⟩ is inserted. The ⟨*options*⟩ are also executed.

Let us now first create a data group. Note that nothing is drawn since the "dummy" data visualization is empty and used only for the definition of the data group.

```
\tikz \datavisualization data group {points} = {
  data {
    x, y
    0, 1
    1, 2
    2, 2
    5, 1
    2, 0
    1, 1
  }
};
```

We can now use this data in different plots:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [school book axes,      visualize as line] data group {points};
\qquad
\tikz \datavisualization [scientific axes=clean, visualize as line] data group {points};
```

\datavisualization … scope[⟨*options*⟩]{⟨*data specification*⟩} …;

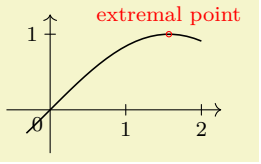
Scopes can be used to nest hierarchical data sets. The ⟨*options*⟩ will be executed with the path `/pgf/data` and will only apply to the data sets specified inside the ⟨*data specification*⟩, which may contain `data` or `scope` commands once more:

```
\datavisualization...
  scope [/data point/experiment=7]
  {
    data [read from file=experiment007-part1.csv]
    data [read from file=experiment007-part2.csv]
    data [read from file=experiment007-part3.csv]
  }
  scope [/data point/experiment=23, format=foo]
  {
    data [read from file=experiment023-part1.foo]
    data [read from file=experiment023-part2.foo]
  };
```

`\datavisualization` … `info[`⟨*options*⟩`]{`⟨*code*⟩`}` … `;`

This command will execute normal TikZ ⟨*code*⟩ at the end of a data visualization. The ⟨*options*⟩ are executed with the normal path `/tikz/`.

The only difference between this command and just giving the ⟨*code*⟩ directly following the data visualization is that inside the ⟨*code*⟩ following an `info` command you still have access to the coordinate system of the data visualization. In sharp contrast, TikZ code given after a data visualization can no longer access this coordinate system.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}[baseline]
  \datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info {
    \draw [red] (visualization cs: x={(.5*pi)}, y=1) circle [radius=1pt]
      node [above,font=\footnotesize] {extremal point};
  };
\end{tikzpicture}
```

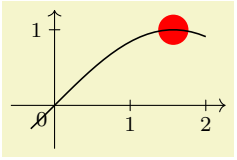
As can be seen, inside a data visualization a special coordinate system is available:

Coordinate system `visualization`

As for other coordinate systems, the syntax is `(visualization cs:`⟨*list of attribute-value pairs*⟩`)`. The effect is the following: For each pair ⟨*attribute*⟩=⟨*value*⟩ in the ⟨*list*⟩ the key `/data point/`⟨*attribute*⟩ is set to ⟨*value*⟩. Then, it is computed where the resulting data point "would lie" on the canvas (however, no data point is passed to the visualizers).

`\datavisualization` … `info'[`⟨*options*⟩`]{`⟨*code*⟩`}` … `;`

This command works like `info`, only the ⟨*code*⟩ will be executed just before the visualization is done. This allows you to draw things *behind* the visualization.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}[baseline]
  \datavisualization [ school book axes, visualize as line ]
  data [format=function] {
    var x : interval [-0.1*pi:2];
    func y = sin(\value x r);
  }
  info' {
    \fill [red] (visualization cs: x={(.5*pi)}, y=1) circle [radius=2mm];
  };
\end{tikzpicture}
```

Predefined node `data visualization bounding box`

This rectangle node stores a bounding box of the data visualization that is currently being constructed. This node can be useful inside `info` commands or when labels need to be added.

Predefined node `data bounding box`

This rectangle node is similar to `data visualization bounding box`, but it keeps track only of the actual data. The spaces needed for grid lines, ticks, axis labels, tick labels, and all other information that is not part of the actual data is not part of this box.

80.7 Advanced: Executing User Code During a Data Visualization

The following keys can be passed to the `\datavisualization` command and allow you to execute some code at some special time during the data visualization process. For details of the process and on which signals are emitted when, see Section 86.

`/tikz/data visualization/before survey=`⟨*code*⟩ (no default)

The ⟨*code*⟩ is passed to the `before survey` method of the data visualization object and then executed at the appropriate time (see Section 86 for details).

The following commands work likewise:

`/tikz/data visualization/at start survey=`⟨*code*⟩ (no default)

`/tikz/data visualization/at end survey=`⟨*code*⟩ (no default)

`/tikz/data visualization/after survey=`⟨*code*⟩ (no default)

`/tikz/data visualization/before visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/at start visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/at end visualization=`⟨*code*⟩ (no default)

`/tikz/data visualization/after visualization=`⟨*code*⟩ (no default)

80.8 Advanced: Creating New Objects

You will need the following key only when you wish to create new rendering pipelines from scratch – instead of modifying an existing pipeline as you would normally do. In the following it is assumed that you are familiar with the concepts of Section 86.

`/tikz/data visualization/new object=`⟨*options*⟩ (no default)

This key serves two purposes:

1. This method makes it easy to create a new object as part of the rendering pipeline, using ⟨*options*⟩ to specify arguments rather that directly calling `\pgfoonew`. Since you have the full power of the keys mechanism at your disposal, it is easy, for instance, to control whether or not parameters to the constructor are expanded or not.
2. The object is not created immediately, but only just before the visualization starts. This allows you to specify that an object must be created, but the parameter values of for its constructor may depend on keys that are not yet set. A typical application is the creating of an axis object: When you say `scientific axes`, the `new object` command is used internally to create two objects representing these axes. However, keys like `x={length=5cm}` can only *later* be used to specify the parameters that need to be passed to the constructor of the objects.

The following keys may be used inside the ⟨*options*⟩:

`/tikz/data visualization/class=`⟨*class name*⟩ (no default)

The class of the to-be-created object.

`/tikz/data visualization/when=`⟨*phase name*⟩ (no default, initially `before survey`)

This key is used to specify when the object is to be created. As described above, the object is not created immediately, but at some time during the rendering process. You can specify any of the phases defined by the data visualization object, see Section 86 for details.

`/tikz/data visualization/store=`⟨*key name*⟩ (no default)

If the ⟨*key name*⟩ is not empty, once the object has been created, a handle to the object will be stored in ⟨*key name*⟩. If a handle is already stored in ⟨*key name*⟩, the object is not created twice.

`/tikz/data visualization/before creation=`⟨*code*⟩ (no default)

This code is executed right before the object is finally created. It can be used to compute values that are then passed to the constructor.

/tikz/data visualization/after creation=⟨*code*⟩ (no default)

This code is executed right after the object has just been created. A handle to the just-created object is available in `\tikzdvobj`.

/tikz/data visualization/arg1=⟨*value*⟩ (no default)

The value to be passed as the first parameter to the constructor. Similarly, the keys `arg2` to `arg8` specify further parameters passed. Naturally, only as many arguments are passed as parameters are set. Here is an example:

```
\tikzdatavisualizationset{
  new object={
    class = example class,
    arg1  = foo,
    arg2  = \bar
  }
}
```

causes the following object creation code to be executed later on:

```
\pgfoonew \tikzdvobj=new example class(foo,\bar)
```

Note that you key mechanisms like `.expand once` to pass the value of a macro instead of the macro itself:

```
\tikzdatavisualizationset{
  new object={
    class              = example class,
    arg1               = foo,
    arg2/.expand once  = \bar
  }
}
```

Now, if `\bar` is set to `This \emph{is} it.` at the moment to object is created later on, the following object creation code is executed:

```
\pgfoonew \tikzdvobj=new example class(foo,This \emph{is} it)
```

/tikz/data visualization/arg1 from key=⟨*key*⟩ (no default)

Works like the `arg1`, only the value that is passed to the constructor is the current value of the specified ⟨*key*⟩ at the moment when the object is created.

```
\tikzdatavisualizationset{
  new object={
    class              = example class,
    arg1 from key      = /tikz/some key
  }
}
\tikzset{some key/.initial=foobar}
```

causes the following to be executed:

```
\pgfoonew \tikzdvobj=new example class(foobar)
```

Naturally, the keys `arg2 from key` to `arg8 from key` are also provided.

/tikz/data visualization/arg1 handle from key=⟨*key*⟩ (no default)

Works like the `arg1 from key`, only the key must store an object and instead of the object a handle to the object is passed to the constructor.

81 Providing Data for a Data Visualization

81.1 Overview

The data visualization system needs a stream of data points as input. These data points can be directly generated by repeatedly calling the `\pgfdatapoint` command, but usually data is available in some special (text) format and one would like to visualize this data. The present section explains how data in some specific format can be fed to the data visualization system.

This section starts with an explanation of the main concepts. Then, the standard formats are listed in the reference section. It is also possible to define new formats, but this an advanced concept which requires an understanding of some of the internals of the parsing mechanism, explained in Section 81.5, and the usage of a rather low-level command, explained in Section 81.6.

81.2 Concepts

For the purposes of this section, let call a *data format* some standardized way of writing down a list of data points. A simple example of a data format is the CSV format (the acronym stands for *comma separated values*), where each line contains a data point, specified by values separated by commas. A different format is the *key–value format*, where data points are specified by lists of key–value pairs. A far more complex format is the PDB-format used by the protein database to describe molecules.

The data visualization system does not use any specific format. Instead, whenever data is read by the data visualization system, you must specify a format parser (or it is chosen automatically for you). It is the job of the parser to read (parse) the data lines and to turn them into data points, that is, to setup appropriate subkeys of `/data point/`.

To give a concrete example, suppose a file contains the following lines:

```
x, y, z
0, 0, 0
1, 1, 0
1, 1, 0.5
0, 1, 0.5
```

This file is in the CSV-format. This format can be read by the `table` parser (which is called thus, rather than "`csv`", since it can also read files in which the columns are separated by, say, a semicolon or a space). The `table` format will then read the data and for each line of the data, except for the headline of course, it will produce one data point. For instance, for the last data point the key `/data point/x` will be set to `0`, the key `/data point/y` will be set to `1`, and the key `/data point/z` will be set to `0.5`.

All parsers are basically line-oriented. This means that, normally, each line in the input data should contain one data point. This rule may not always apply, for instance empty lines are typically ignored and sometimes a data point may span several lines, but deviating from this "one data point per line" rule makes parsers harder to program.

81.3 Reference: Built-In Formats

The following format is the default format, when no `format=...` is specified.

Format `table`

This format is used to parse data that is formatted in the following manner: Basically, each line consists of *values* that are separated by a *separator* like a comma or a space. The values are stored in different *attributes*, that is, subkeys of `/data point` like `/data point/x`. In order to decide which attribute is chosen for a give value, the headline is important. This is the first non-empty line of a table. It is formatted in the same way as normal data lines (value separated by the separator), but the meaning of the values is different: The first value in the headline is the name of the attribute where the first values in the following lines should go each time. Similarly, the second value in the headline is the name of the attribute for the second values in the following lines, and so on.

A simple example is the following:

```
angle, radius
0, 1
45, 2
90, 3
135, 4
```

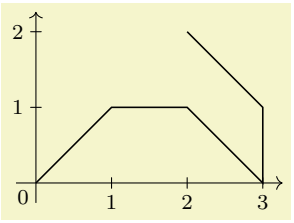
The headline states that the values in the first column should be stored in the `angle` attribute (`/data point/angle` to be precise) and that the values in the second column should be stored in the `radius` attribute. There are four data points in this data set.

The format will tolerate too few or too many values in a line. If there are less values in a line than in the headline, the last attributes will simply be empty. If there are more values in a line than in the headline, the values are stored in attributes called `/data point/attribute` ⟨*column number*⟩, where the first value of a line gets ⟨*column number*⟩ equal to `1` and so on.

The `table` format can be configured using the following options:

`/pgf/data/separator`=⟨*character*⟩ (no default, initially `,`)

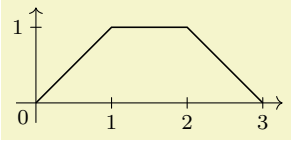
Use this key to change which character is used to separate values in the headline and in the data lines. To set the separator to a space, either set this key to an empty value or say `separator=\space`. Note that you must surround a comma by curly braces if you which to (re)set the separator character to a space.




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
    data [separator=\space] {
      x y
      0 0
      1 1
      2 1
      3 0
    }
    data [separator=;] {
      x; y; z
      3; 1; 0
      2; 2; 0
    };
\end{tikzpicture}
```

`/pgf/data/headline`=⟨*headline*⟩ (no default)

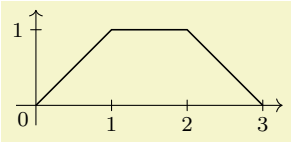
When this key is set to a non-empty value, the value of ⟨*headline*⟩ is used as the headline and the first line of the data is treated as a normal line rather than as a headline.




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
    data [headline={x, y}] {
      0, 0
      1, 1
      2, 1
      3, 0
    };
\end{tikzpicture}
```

Format `named`

Basically, each line of the data must consist of a comma-separated sequence of attribute–values pairs like `x=5, lo=500`. This will cause the attribute `/data point/x` to be set to `5` and `/data point/lo` to be set to `500`.

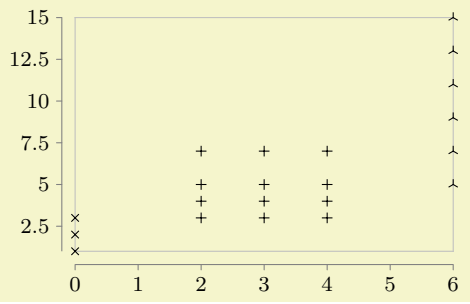



```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
    data [format=named] {
      x=0, y=0
      x=1, y=1
      x=2, y=1
      x=3, y=0
    };
\end{tikzpicture}
```

However, instead of just specifying a single value for an attribute as in `x=5`, you may also specify a whole set of values as in `x={1,2,3}`. In this case, three data points will be created, one for each value

in the list. Indeed, the `\foreach` statement is used to iterate over the list of values, so you can write things like `x={1,...,5}`.

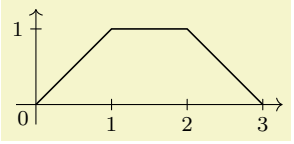
It is also permissible to specify lists of values for more than one attribute. In this case, a data point is created for each possible combination of values in the different lists:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
  [scientific axes=clean,
   visualize as scatter/.list={a,b,c},
   style sheet=cross marks]
data [format=named] {
  x=0,       y={1,2,3},       set=a
  x={2,3,4}, y={3,4,5,7},     set=b
  x=6,       y={5,7,...,15},  set=c
};
```

Format `TeX code`

This format will simply execute each line of the data, each of which should contain some normal TEX code. Note that at the end of each line control returns to the format handler, so for instance the arguments of a command may not be spread over several lines. However, not each line needs to produce a data point.




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as line]
    data [format=TeX code] {
      \pgfkeys{/data point/.cd,x=0, y=0} \pgfdatapoint
      \pgfkeys{/data point/.cd,x=1, y=1} \pgfdatapoint
      \pgfkeys{/data point/x=2}          \pgfdatapoint
      \pgfkeyssetvalue{/data point/x}{3}
      \pgfkeyssetvalue{/data point/y}{0} \pgfdatapoint
    };
\end{tikzpicture}
```

81.4 Reference: Advanced Formats

TikZ Library `datavisualization.formats.functions`

```
\usetikzlibrary{datavisualization.formats.functions} % LaTeX and plain TeX
\usetikzlibrary[datavisualization.formats.functions] % ConTeXt
```

This library defines the formats described in the following, which allow you to specify the data points indirectly, namely via a to-be-evaluated function.

Format `function`

This format allows you to specify a function that is then evaluated in order to create the desired data points. In other words, the data lines do not contain the data itself, but rather a functional description of the data.

The format used to specify the function works as follows: Each nonempty line of the data should contain at least one of either a *variable declaration* or a *function declaration*. A variable declaration signals that a certain attribute will range over a given interval. The function declarations will then, later, be evaluated for values inside this interval. The syntax for a variable declaration is one of the following:

1. `var` ⟨*variable*⟩ `: interval[`⟨*low*⟩`:`⟨*high*⟩`] samples` ⟨*number*⟩`;`
2. `var` ⟨*variable*⟩ `: interval[`⟨*low*⟩`:`⟨*high*⟩`] step` ⟨*step*⟩`;`
3. `var` ⟨*variable*⟩ `: {`⟨*values*⟩`};`

In the first case, if the optional `samples` part is missing, the number of `samples` is taken from the value stored in the following key:

`/pgf/data/samples=`⟨*number*⟩ (no default, initially `25`)

Sets the number of samples to be used when no sample number is specified.

The meaning of declaring a variable declaration to range over an `interval` is that the attribute named ⟨*variable*⟩, that is, the key `/data point/`⟨*variable*⟩, will range over the interval [⟨*low*⟩, ⟨*high*⟩]. If the number of `samples` is given (directly or indirectly), the interval is evenly divided into ⟨*number*⟩ many points and the attribute is set to each of these values. Similarly, when a ⟨*step*⟩ is specified, this stepping is used to increase ⟨*low*⟩ iteratively up to the largest value that is still less or equal to ⟨*high*⟩.

The meaning of declaring a variable using a list of ⟨*values*⟩ is that the variable will simply iterate over the values using `\foreach`.

You can specify more than one variable. In this case, each variable is varied independently of the other variables. For instance, if you declare an x-variable to range over the interval $[0, 1]$ in 25 steps and you also declare a y-variable to range over the same interval, you get a total of 625 value pairs.

The variable declarations specify which (input) variables will take which values. It is the job of the *function declarations* to specify how some additional attributes are to be computed. The syntax of a function declaration is as follows:

`func` ⟨*attribute*⟩ `=` ⟨*expression*⟩`;`

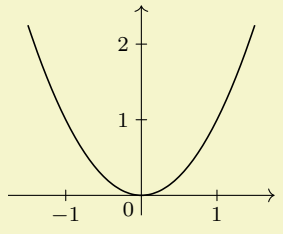
The meaning of such a declaration is the following: For each setting of the input variables (the variables specified using the `var` declaration), evaluate the ⟨*expression*⟩ using the standard mathematical parser of Ti*k*Z. The resulting value is then stored in `/data point/`⟨*attribute*⟩.

Inside ⟨*expression*⟩ you can reference data point attributes using the following command, which is only defined inside such an expression:

`\value{`⟨*variable*⟩`}`

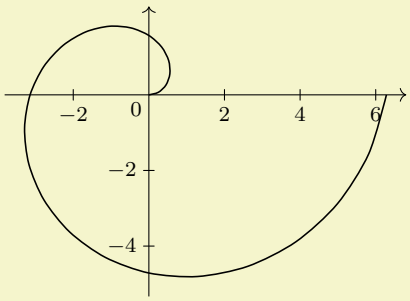
This expands to the current value of the key `/data point/`⟨*variable*⟩.

There can be multiple function declarations in a single data specification. In this case, all of these functions will be evaluated for each setting of input variables.



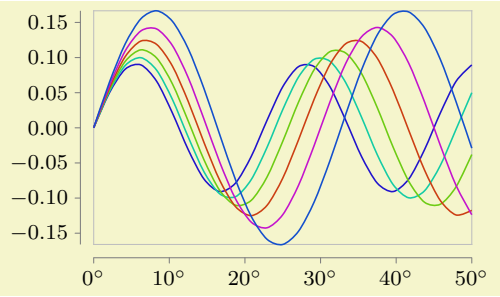

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz
  \datavisualization [school book axes, visualize as smooth line]
    data [format=function] {
      var x : interval [-1.5:1.5];

      func y = \value x * \value x;
    };
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  all axes={unit length=5mm, ticks={step=2}},
  visualize as smooth line]
data [format=function] {
  var t : interval [0:2*pi];

  func x = \value t * cos(\value t r);
  func y = \value t * sin(\value t r);
};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean,
  y axis={ticks={style={
        /pgf/number format/fixed,
        /pgf/number format/fixed zerofill,
        /pgf/number format/precision=2}}},
  x axis={ticks={tick suffix=${}^\circ$}},
  visualize as smooth line/.list={1,2,3,4,5,6},
  style sheet=vary hue]
data [format=function] {
  var set : {1,...,6};
  var x : interval [0:50];
  func y = sin(\value x * (\value{set}+10))/(\value{set}+5);
};
```

81.5 Advanced: The Data Parsing Process

Whenever data is fed to the data visualization system, it will be handled by the `\pgfdata` command, declared in the `datavisualization` module. The command is both used to parse data stored in external sources (that is, in external files or which is produced on the fly by calling an external command) as well as data given inline. A data format does not need to know whether data comes from a file or is given inline, the `\pgfdata` command will take care of this.

Since TEX will always read files in a line-wise fashion, data is always fed to data format parsers in such a fashion. Thus, even it would make more sense for a format to ignore line-breaks, the parser must still handle data given line-by-line.

Let us now have a look at how `\pgfdata` works.

`\pgfdata[`⟨*options*⟩`]{`⟨*inline data*⟩`}`

This command is used to feed data to the visualization pipeline. This command can only be used when a data visualization object has been properly setup, see Section 80.

Basic options. The `\pgfdata` command may be followed by ⟨*options*⟩, which are executed with the path `/pgf/data/`. Depending on these options, the ⟨*options*⟩ may either be followed by ⟨*inline data*⟩ or, alternatively, no ⟨*inline data*⟩ is present and the data is read from an external source.

The first important option is ⟨*source*⟩, which governs which of these two alternatives applies:

`/pgf/data/read from file=`⟨*filename*⟩ (no default, initially empty)

If you set the `read from file` attribute to a non-empty ⟨*filename*⟩, the data will be read from this file. In this case, no ⟨*inline data*⟩ may be present, not even empty curly braces should be provided. If `read from file` is empty, the data must directly follow as ⟨*inline data*⟩.

```
% Data is read from two external files:
\pgfdata[format=table, read from file=file1.csv]
\pgfdata[format=table, read from file=file2.csv]
```

```
% Data is given inline:
\pgfdata[format=table]
{
  x, y
  1, 2
  2, 3
}
```

`/pgf/data/inline` (no value)

This is a shorthand file `read from file={}`. You can add this to make it clear(er) to the reader that data follows inline.

The second important key is `format`, which is used to specify the data format:

`/pgf/data/format=`⟨*format*⟩ (no default, initially `table`)

Use this key to locally set the format used for parsing the data. The ⟨*format*⟩ must be a format that has been previously declared using the `\pgfdeclaredataformat` command. See the reference section for a list of the predefined formats.

In case all your data is in a certain format, you may wish to generally set the above key somewhere at the beginning of your file. Alternatively, you can use the following style to setup the `format` key and possibly further keys concerning the data format:

`/pgf/every data` (style, no value)

This style is executed by `\pgfdata` before the ⟨*options*⟩ are parsed.

Note that the path of this key is just `/pgf/`, not `/pgf/data/`. Also note that Ti*k*Z internally sets the value of this key up in such a way that the keys `/tikz/every data` and also `/tikz/data visualization/every data` are executed. The bottom line of this is that when using Ti*k*Z, you should not set this key directly, set `/tikz/every data` instead.

Gathering of the data. Once the data format and the source have been decided upon, the data is "gathered". During this phase the data is not actually parsed in detail, but just gathered so that it can later be parsed during the visualization. There are two different ways in which the data is gathered:

- In case you have specified an external source, the data visualization object is told (by means of invoking the `add data` method) that it should (later) read data from the file specified by the `source` key using the format specified by the `format` key. The file is not read at this point, but only later during the actual visualization.
- Otherwise, namely when data is given inline, depending on which format is used, some catcodes get changed. This is necessary since TEX's special characters are often not-so-special in a certain format.

  Independently of the format, the end-of-line character (carriage return) is made an active character.

  Finally, the ⟨*inline data*⟩ is then read as a normal argument and the data visualization object is told that later on it should parse this data using the given format parser. Note that in this case the data visualization object must store the whole data internally.

In both cases the "data visualization object" is the object stored in the `/pgf/data visualization/obj` key.

Parsing of the data. During the actual data visualization, all code that has been added to the data visualization object by means of the `add data` method is executed several times. It is the job of this code to call the `\pgfdatapoint` method for all data points present in the data.

When the `\pgfdata` method calls `add data`, the code that is passed to the data visualization object is just a call to internal macros of `\pgfdata`, which are able to parse the data stored in an external file or in the inlined data. Independently of where the data is stored, these macros always do the following:

1. The catcodes are setup according to what the data format requires.
2. Format-specific startup code gets called, which can initialize internal variables of the parsing process. (The catcode changes are not part of the startup code since in order to read inline data `\pgfdata` must be able to setup to temporarily setup the catcodes needed later on by the parsers, but since no reading is to be done, no startup code should be called at this point.)
3. For each line of the data a format-specific code handler, which depends on the data format, is called. This handler gets the current line as input and should call `\pgfdatapoint` once for each data point that is encoded by this line (a line might define multiple data points or none at all). Empty lines are handled by special format-specific code.
4. At the end, format-specific end code is executed.

For an example of how this works, see the description of the `\pgfdeclaredataformat` command.

Data sets. There are three options that allow you to create *data sets*. Such a data set is essentially a macro that stores a pre-parsed set of data that can be used multiple times in subsequent visualizations (or even in the same visualization).

`/pgf/data/new set=`⟨*name*⟩ (no default)

Creates an empty data set called ⟨*name*⟩. If a data set of the same name already exists, it is overwritten and made empty. Data sets are global.

/pgf/data/store in set=⟨*name*⟩ (no default)

When this key is set to any non-empty ⟨*name*⟩ and if this ⟨*name*⟩ has previously been used with the `new set` key, then the following happens: For the current `\pgfdata` command, all parsed data is not passed to the rendering pipeline. Instead, the parsed data is appended to the data set ⟨*name*⟩. This includes all options parsed to the `\pgfdata` command, which is why neither this key nor the previous key should be passed as options to a `\pgfdata` command.

/pgf/data/use set=⟨*name*⟩ (no default)

This works similar to `read from file`. When this key is used with a `\pgfdata` command, no inline data may follow. Instead, the data stored in the data set ⟨*name*⟩ is used.

81.6 Advanced: Defining New Formats

In order to define a new data format you can use the following command, which is basic layer command defined in the module `datavisualization`:

`\pgfdeclaredataformat{`⟨*format name*⟩`}{`⟨*catcode code*⟩`}{`⟨*startup code*⟩`}{`⟨*line arguments*⟩`}{`⟨*line code*⟩`}{`⟨*empty line code*⟩`}{`⟨*end code*⟩`}`

This command defines a new data format called ⟨*format name*⟩, which can subsequently be used in the `\pgfdata` command. (The Ti*k*Z's `data` maps directly to `\pgfdata`, so the following applies to Ti*k*Z as well.)

As explained in the description of the `\pgfdata` command, when data is being parsed that is formatted according to ⟨*format name*⟩, the following happens:

1. The ⟨*catcode code*⟩ is executed. This code should just contain catcode changes. The ⟨*catcode code*⟩ will also be executed when inline data is read.
2. Next, the ⟨*startup code*⟩ is executed.
3. Next, for each non-empty line of the data, the line is passed to a macro whose argument list is given by ⟨*line arguments*⟩ and whose body is given by ⟨*line code*⟩. The idea is that you can use TEX's powerful pattern matching capabilities to parse the non-empty lines. See also the below example.
4. Empty lines are not processed by the ⟨*line code*⟩, but rather by the ⟨*empty line code*⟩. Typically, empty lines can simply be ignored and in this case you can let this parameter be empty.
5. At the end of the data, the ⟨*end code*⟩ is executed.

As an example, let us now define a simple data format for reading files formatted in the following manner: Each line should contain a coordinate pair as in `(1.2,3.2)`, so two numbers separated by a comma and surrounded by parentheses. To make things more interesting, suppose that the hash mark symbol can be used to indicate comments. Here is an example of some data given in this format:

```
# This is some data formatted according to the "coordinates" format
(0,0)
(0.5,0.25)
(1,1)
(1.5,2.25)
(2,4)
```

A format parser for this format could be defined as follows:

```
\pgfdeclaredataformat{coordinates}
% First comes the catcode argument. We turn the hash mark into a comment character.
{\catcode`\#=14\relax}
% Second comes the startup code. Since we do not need to setup things, we can leave
% it empty. Note that we could also set it to something like \begingroup, provided we
% put an \endgroup in the end code
{}
% Now comes the arguments for non-empty lines. Well, these should be of the form
% (#1,#2), so we specify that:
{(#1,#2)}
% Now we must do something with a line of this form. We store the #1 argument in
% /data point/x and #2 in /data point/y. Then we call \pgfdatapoint to create a data point.
{
  \pgfkeyssetvalue{/data point/x}{#1}
  \pgfkeyssetvalue{/data point/y}{#2}
  \pgfdatapoint
}
% We ignore empty lines:
{}
% And we also have no end-of-line code.
{}
```

This format could now be used as follows:

```
\begin{tikzpicture}
  \datavisualization[school book axes, visualize as smooth line]
  data [format=coordinates] {
    # This is some data formatted according
    # to the "coordinates" format
    (0,0)
    (0.5,0.25)
    (1,1)
    (1.5,2.25)
    (2,4)
  };
\end{tikzpicture}
```

82 Axes

82.1 Overview

When a data point is visualized, the most obvious way of creating a visual representation of its many attributes is to vary *where* the data point is shown. The data visualization system uses *axes* to turn data point attributes into positions on a page. The simplest – and most common – use of axes is to vary the horizontal position of data points according to one attribute and to vary the vertical position according to another attribute. In contrast, in a polar plot one attribute dictates the distance of the data point from the origin and another attribute describes the angle. From the data visualization engine's point of view, in both cases two *axes* are involved.

In addition to specifying how the value of a certain attribute is converted into a displacement on the page, an axis is also typically (but not always) visualized ("drawn") somewhere on the page. In this case, it is also customary to add a visual representation on this axis of which attribute values correspond to which positions on the page – something commonly known as *ticks*. Similar to ticks, *grid lines* also indicate positions where a certain attribute has a certain value, but instead of just indicating a single position on an axis, a grid line goes through all points that share an attribute value.

In the following, in Section 82.2 we first have a look at how axes can be defined and configured. As you will see, a lot of powerful configurations are available, but you will rarely define and configure an axis from scratch. Rather, it is more common to use a preconfigured axis instead. Section 82.3 introduces *axis systems*, which are predefined bundles of axes. You can define your own axis systems, but, again, in most cases it will suffice to just use one of the many preconfigured axis systems and use a few options to configure it so that it fits your need. Section 82.4 explains how ticks and grid lines can be configured. Again, several layers of options allow you to configure the way ticks look and where they are placed in great detail.

This section documents the standard axis systems that are always available. For polar axis systems, a special library needs to be loaded, which is documented in Section 85.

82.2 Basic Configuration of Axes

Inside the data visualization system, an *axis* is roughly a "systematic, named way of mapping an attribute to a position on a page". For instance, the classical "x-axis" is the "systematic way of mapping the value of the `x` attribute of data points to a horizontal position on the page". An axis is *not* its visual representation (such as the horizontal line with the ticks drawn to represent the x-axis), but a visual representation can be created once an axis has been defined.

The transformation of an attribute value (such as the value `1000000000` for the `x` attribute) to a specific displacement of the corresponding data point on the page involves two steps:

1. First, the range of possible values such as $[-5.6 \cdot 10^{12}, 7.8 \cdot 10^{12}]$ must be mapped to a "reasonable" interval such as $[0\text{cm}, 5\text{cm}]$ or $[0°, 180°]$. Ti*k*Z's drawing routines will only be able to cope with values from such a "reasonable" interval.

2. Second, the values from the reasonable interval must be mapped to a transformation.

The first step is always the same for all axes, while the second requires different strategies. For this reason, the command `new axis base` is used to create a "basic" axis that has a "scaling mapper", whose job it is to map the range of values of a specific attribute to a reasonable interval, but such a basic axis does not define an actual transformation object. For this second step, additional objects such as a `linear transformer` need to be created separately.

82.2.1 Usage

To create an axis, the key `new axis base` is used first. Since this key does not create a transformation object, users typically do not use this key directly. Rather, it is used internally by other keys that create "real" axes. These keys are listed in Section 82.2.8.

`/tikz/data visualization/new axis base=`⟨*axis name*⟩ (no default)

This key defines a new axis for the current data visualization called ⟨*name*⟩. This has two effects:

1. A so called *scaling mapper* is created that will monitor a certain attribute, rescale it, and map it to another attribute. (This will be explained in detail in a moment.)
2. The ⟨*axis name*⟩ is made available as a key that can be used to configure the axis:

`/tikz/data visualization/`⟨*axis name*⟩`=`⟨*options*⟩ (no default)

This key becomes available once `new axis base=`metaaxis name has been called. It will execute the ⟨*options*⟩ with the path prefix `/tikz/data visualization/axis options`.

```
[new axis base=my axis,
 my axis={attribute=some attribute}]
```

3. The ⟨*axis name*⟩ becomes part of the current set of axes. This set can be accessed through the following key:

   `/tikz/data visualization/all axes=`⟨*options*⟩ (no default)

   This key passes the ⟨*options*⟩ to all axes inside the current scope, just as if you had written ⟨*some axis name*⟩`=`⟨*options*⟩ for each ⟨*some axis name*⟩ in the current scope, including the just-created name ⟨*axis name*⟩.

There are many ⟨*options*⟩ that can be passed to a newly created axis. They are explained in the rest of this section.

Note the `new axis base` does *not* cause attributes to be mapped to positions on a page. Rather, special keys like `new Cartesian axis` first use `new axis base` to create an axis and then create an internal object that performs a linear mapping of the attribute to positions along a vectors.

82.2.2 The Axis Attribute

The first main job of an axis is to map the different values of some attribute to a reasonable interval. To achieve this, the following options are important (recall that these options are passed to the key whose name is the name of the axis):

`/tikz/data visualization/axis options/attribute=`⟨*attribute*⟩ (no default)

Specifies that the axis is used to transform the data points according the different values of the key `/data point/`⟨*attribute*⟩. For instance, when we create a classical two-dimensional Cartesian coordinate system, then there are two axes called `x axis` and `y axis` that monitor the values of the attributes `/data point/x` and `/data point/y`, respectively:

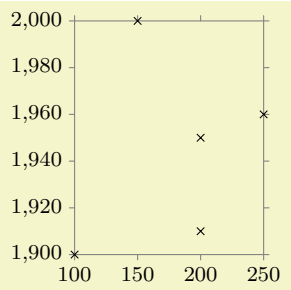
```
[new axis base=x axis,
 new axis base=y axis,
 x axis={attribute=x},
 y axis={attribute=y}]
```

In another example, we also create an `x axis` and a `y axis`. However, this time, we want to plot the values of the `/data point/time` attribute on the x-axis and, say, the value of the `height` attribute on the y-axis:

```
[new axis base=x axis,
 new axis base=y axis,
 x axis={attribute=time},
 y axis={attribute=height}]
```

During the data visualization, the ⟨*attribute*⟩ will be "monitored" during the survey phase. This means that for each data point, the current value of `/data point/`⟨*attribute*⟩ is examined and the minimum value of all of these values as well as the maximum value is recorded internally. Note that this works even when very large numbers like `100000000000` are involved.

Here is a real-life example. The `scientific axes` create two axes, called `x axis` and `y axis`, respectively.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [scientific axes,
                          x axis={attribute=people, length=2.5cm, ticks=few},
                          y axis={attribute=year},
                          visualize as scatter]
  data {
    year, people
    1900, 100
    1910, 200
    1950, 200
    1960, 250
    2000, 150
  };
```

82.2.3 The Axis Attribute Range Interval

Once an attribute has been specified for an axis, the data visualization engine will start monitoring this value. This means that before anything actual visualization is done, a "survey phase" is used to determine the range of values encountered for the attribute for all data points. This range of values results in what is called the *attribute range interval*. Its minimum is the smallest value encountered in the data and its maximum is the largest value.

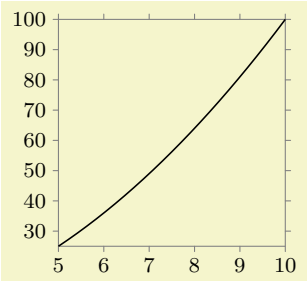
Even though the attribute range interval is computed automatically and even though you typically do not need to worry about it, there are some situations where you may wish to set or enlarge the attribute range interval:

- You may wish to start the interval with 0, even though the range of values contains only positive values.
- You may wish to slightly enlarge the interval so that, say, the maximum is some "nice" value like `100` or `60`.

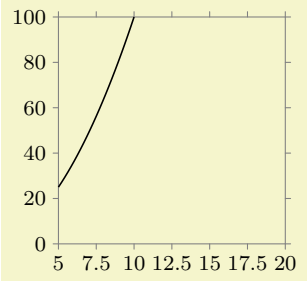
The following keys can be used to influence the size of the attribute range interval:

`/tikz/data visualization/axis options/include value=`⟨*list of value*⟩ (no default)

This key "fakes" data points for which the attribute's values are in the comma-separated ⟨*list of values*⟩. For instance, when you write `include value=0`, then the attribute range interval is guaranteed to contain `0` – even if the actual data points are all positive or all negative.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes, all axes={length=3cm},
                          visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes, all axes={length=3cm},
                          visualize as line,
                          x axis={include value=20},
                          y axis={include value=0}]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

`/tikz/data visualization/axis options/min value=`⟨*value*⟩ (no default)

This key allows you to simply set the minimum value, regardless of which values are present in the actual data. This key should be used with care: If there are data points for which the attribute's value is less than ⟨*value*⟩, they will still be depicted, but typically outside the normal visualization area. Usually, saying `include value=`⟨*value*⟩ will achieve the same as saying `min value=`⟨*value*⟩, but with less danger of creating ill-formed visualizations.

`/tikz/data visualization/axis options/max value=`⟨*value*⟩ (no default)

Works like `min value`.

82.2.4 Scaling: The General Mechanism

The above key allows us specify which attribute should be "monitored". The next key is used to specify what should happen with the observed values.

`/tikz/data visualization/axis options/scaling=`⟨*scaling spec*⟩ (no default)

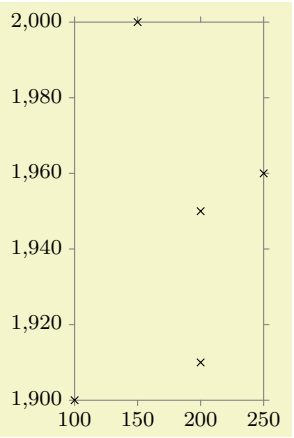
The ⟨*scaling spec*⟩ must have the following form:

⟨s_1⟩ `at` ⟨t_1⟩ `and` ⟨s_2⟩ `at` ⟨t_2⟩

This means that monitored values in the interval $[s_1, s_2]$ should be mapped to values the "reasonable" interval $[t_1, t_2]$, instead. For instance, we might write

```
[y axis = {scaling = 1900 at 0cm and 2000 at 5cm}]
```

in order to map dates between 1900 and 2000 to the dimension interval [0cm, 5cm].




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
   [scientific axes,
    x axis={attribute=people, length=2.5cm, ticks=few},
    y axis={attribute=year, scaling=1900 at 0cm and 2000 at 5cm},
    visualize as scatter]
  data {
    year, people
    1900, 100
    1910, 200
    1950, 200
    1960, 250
    2000, 150
  };
```

So much for the basic idea. Let us now have a detailed look at what happens.

Number format and the min and max keywords. The source values s_1 and s_2 are typically just numbers like `3.14` or `10000000000`. However, as described in Section 80.2, you can also specify expressions like `(pi/2)`, provided that (currently) you put them in parentheses.

Instead of a number, you may alternatively also use the two key words `min` and `max` for s_1 and/or s_2. In this case, `min` evaluates to the smallest value observed for the attribute in the data, symmetrically `max` evaluates to the largest values. For instance, in the above example with the `year` attribute ranging from `1900` to `2000`, the keyword `min` would stand for `1900` and `max` for `2000`. Similarly, for the `people` attribute `min` stands for `100` and `max` for `250`. Note that `min` and `max` can only be used for s_1 and s_2, not for t_1 and t_2.

A typical use of the `min` and `max` keywords is to say

```
scaling = min at 0cm and max at 5cm
```

to map the complete range of values into an interval of length of 5cm.

The interval $[s_1, s_2]$ need not contain all values that the ⟨*attribute*⟩ may attain. It is permissible that values are less than s_1 or more than s_2.

Linear transformation of the attribute. As indicated earlier, the main job of an axis is to map values from a "large" interval $[s_1, s_2]$ to a more reasonable interval $[t_1, t_2]$. Suppose that for the current data point the value of the key `/data point/`⟨*attribute*⟩ is the number v. In the simplest case, the following happens: A new value v' is computed so that $v' = t_1$ when $v = s_1$ and $v' = t_2$ when $v = s_2$ and v' is some value in between t_1 and t_2 then v is some value in between s_1 and s_2. (Formally, in this basic case $v' = t_1 + (v - s_1)\frac{t_2 - t_1}{s_2 - s_1}$.)

Once v' has been computed, it is stored in the key `/data point/`⟨*attribute*⟩`/scaled`. Thus, the "reasonable" value v' does not replace the value of the attribute, but it is placed in a different key. This means that both the original value and the more "scaled" values are available when the data point is visualized.

As an example, suppose you have written

```
[x axis = {attribute = x, scaling=1000 at 20 and 2000 at 30}]
```

Now suppose that `/data point/x` equals `1200` for a data point. Then the key `/data point/x/scaled` will be set to `22` when the data point is being visualized.

Nonlinear transformations of the attribute. By default, the transformation of $[s_1, s_2]$ to $[t_1, t_2]$ is the linear transformation described above. However, in some case you may be interested in a different kind of transformation: For example, in a logarithmic plot, values of an attribute may range between, say, `1` and `1000` and we want an axis of length `3cm`. So, we would write

```
[x axis = {attribute = x, scaling=1 at 0cm and 1000 at 3cm}]
```

Indeed, `1` will now be mapped to position `0cm` and `1000` will be mapped to position `3cm`. Now, the value `10` will be mapped to approximately `0.03cm` because it is (almost) at one percent between `1` and `1000`. However, in a logarithmic plot we actually want `10` to be mapped to the position `1cm` rather than `0.03cm` and we want `100` to be mapped to the position `2cm`. Such a mapping a *nonlinear* mapping between the intervals.

In order to achieve such a nonlinear mapping, the `function` key can be used, whose syntax is described in a moment. The effect of this key is to specify a function $f\colon \mathbb{R} \to \mathbb{R}$ like, say, the logarithm function. When such a function is specified, the mapping of v to v' is computed as follows:

$$v' = t_1 + (f(s_2) - f(v))\frac{t_2 - t_1}{f(s_2) - f(s_1)}.$$

The syntax of the `function` key is described next, but you typically will not call this key directly. Rather, you will use a key like `logarithmic` that installs appropriate code for the `function` key for you.

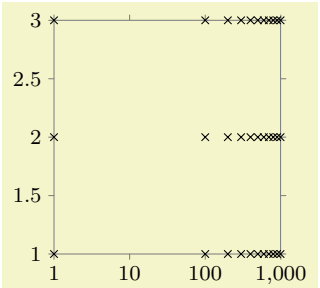
`/tikz/data visualization/axis options/function=`⟨*code*⟩ (no default)

The ⟨*code*⟩ should specify a function f that is applied during the transformation of the interval $[s_1, s_2]$ to the interval $[t_1, t_2]$ in the following way: When the ⟨*code*⟩ is called, the macro `\pgfvalue` will have been set to an internal representation of the to-be-transformed value v. You can then call the commands of the math-micro-kernel of the data visualization system, see Section 86.4, to compute a new value. This new value must once more be stored in `\pgfvalue`.

The most common use of this key is to say

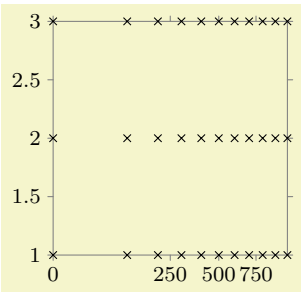
```
some axis={function=\pgfdvmathln{\pgfvalue}{\pgfvalue}}
```

This specifies that the function f is the logarithm function.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
   [scientific axes,
    x axis={ticks={major={at={1,10,100,1000}}},
            scaling=1 at 0cm and 1000 at 3cm,
            function=\pgfdvmathln{\pgfvalue}{\pgfvalue}},
    visualize as scatter]
  data [format=named] {
    x={1,100,...,1000}, y={1,2,3}
  };
```

Another possibility might be to use the square-root function, instead:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
   [scientific axes,
    x axis={ticks=few,
            scaling=1 at 0cm and 1000 at 3cm,
            function=\pgfdvmathunaryop{\pgfvalue}{sqrt}{\pgfvalue}},
    visualize as scatter]
  data [format=named] {
    x={0,100,...,1000}, y={1,2,3}
  };
```

Default scaling. When no scaling is specified, it may seem natural to use $[0, 1]$ both as the source and the target interval. However, this would not work when the logarithm function is used as transformations: In this case the logarithm of zero would be computed, leading to an error. Indeed, for a logarithmic axis it is far more natural to use $[1, 10]$ as the source interval and $[0, 1]$ as the target interval.

For these reasons, the default value for the `scaling` that is used when no value is specified explicitly can be set using a special key:

`/tikz/data visualization/axis options/scaling/default`=⟨*text*⟩ (no default)

The ⟨*text*⟩ is used as `scaling` whenever no other scaling is specified. This key is mainly used when a transformation function is set using `function`; normally, you will not use this key directly.

Most of the time, you will not use neither the `scaling` nor the `function` key directly, but rather you will use one of the following predefined styles documented in the following.

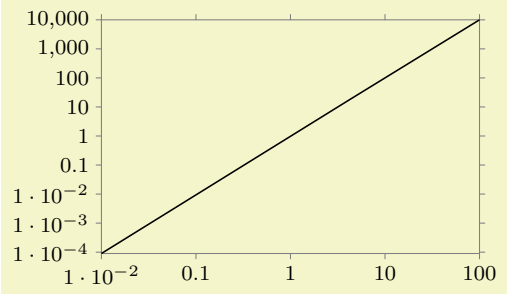
82.2.5 Scaling: Logarithmic Axes

`/tikz/data visualization/axis options/logarithmic` (no value)

When this key is used with an axis, three things happen:

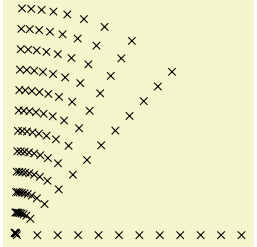
1. The transformation `function` of the axis is setup to the logarithm.
2. The strategy for automatically generating ticks and grid lines is set to the `exponential strategy`, see Section 82.4.13 for details.
3. The default scaling is setup sensibly.

All told, to turn an axis into a logarithmic axis, you just need to add this option to the axis.

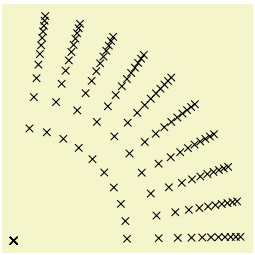



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          x axis={logarithmic},
                          y axis={logarithmic},
                          visualize as line]
 data [format=function] {
   var x : interval [0.01:100];
   func y = \value x * \value x;
 };
```

Note that this will work with any axis, including, say, the degrees on a polar axis:



```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes,
     angle axis={logarithmic, scaling=1 at 0 and 90 at 90},
     radius axis={scaling=0 at 0cm and 100 at 3cm},
     visualize as scatter]
  data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
  };
```

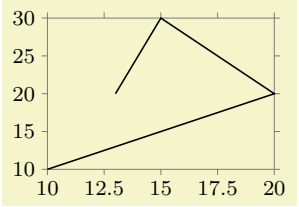


```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes,
     angle axis={degrees},
     radius axis={logarithmic, scaling=1 at 0cm and 100 at 3cm},
     visualize as scatter]
  data [format=named] {
    angle={1,10,...,90}, radius={1,10,...,100}
  };
```

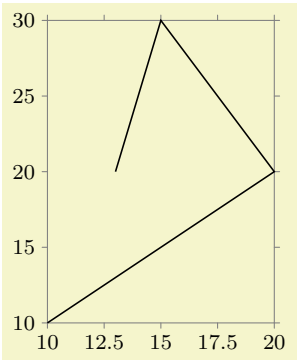
82.2.6 Scaling: Setting the Length or Unit Length

`/tikz/data visualization/axis options/length`=⟨*dimension*⟩ (no default)

Sets `scaling` to `min at 0cm and max at` ⟨*dimension*⟩. The effect is that the range of all values of the axis's attribute will be mapped to an interval of exact length ⟨*dimension*⟩.



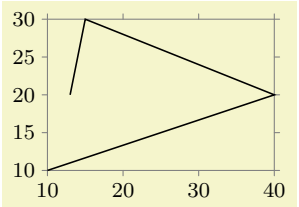

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [scientific axes,
                          x axis={length=3cm},
                          y axis={length=2cm},
                          all axes={ticks=few},
                          visualize as line]
    data {
      x, y
      10, 10
      20, 20
      15, 30
      13, 20
    };
```




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [scientific axes,
                          x axis={length=3cm},
                          y axis={length=4cm},
                          all axes={ticks=few},
                          visualize as line]
    data {
      x, y
      10, 10
      20, 20
      15, 30
      13, 20
    };
```

`/tikz/data visualization/axis options/unit length`=⟨*dimension*⟩ `per` ⟨*number*⟩ `units` (no default)

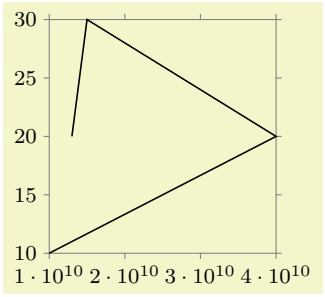
Sets `scaling` to `0 at 0cm and 1 at` ⟨*dimension*⟩. In other words, this key allows you to specify how long a single unit should be. This key is particularly useful when you wish to ensure that the same scaling is used across multiple axes or pictures.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [scientific axes,
                          all axes={ticks=few, unit length=1mm},
                          visualize as line]
    data {
      x, y
      10, 10
      40, 20
      15, 30
      13, 20
    };
```

The optional `per` ⟨*number*⟩ `units` allows you to apply more drastic scaling. Suppose that you want to plot a graph where one billion corresponds to one centimeter. Then the unit length would be need to

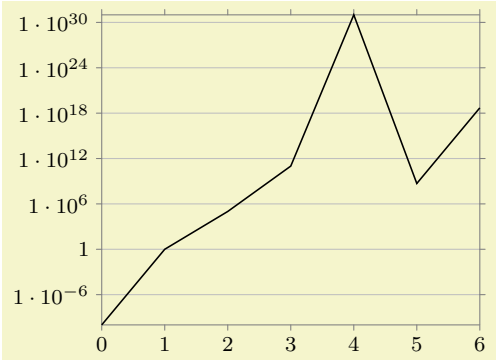
be set to a hundredth of a nanometer – much too small for TEX to handle as a dimension. In this case, you can write `unit length=1cm per 1000000000 units`:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
  [scientific axes,
   x axis={unit length=1mm per 1000000000 units, ticks=few},
   visualize as line]
 data {
   x, y
   10000000000, 10
   40000000000, 20
   15000000000, 30
   13000000000, 20
 };
```

`/tikz/data visualization/axis options/power unit length=`⟨*dimension*⟩ (no default)

This key is used in conjunction with the `logarithmic` setting. It cases the `scaling` to be set to `1 at 0cm and 10 at` ⟨*dimension*⟩. This causes a "power unit", that is, one power of ten in a logarithmic plot, to get a length of ⟨*dimension*⟩. Again, this key is useful for ensuring that the same scaling is used across multiple axes or pictures.



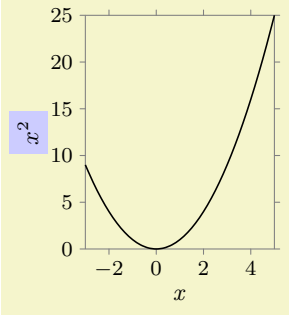

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
  [scientific axes,
   y axis={logarithmic, power unit length=1mm, grid},
   visualize as line]
 data {
   x, y
   0, 0.0000000001
   1, 1
   2, 100000
   3, 100000000000
   4, 10000000000000000000000000000000
   5, 500000000
   6, 5000000000000000000
 };
```

82.2.7 Axis Label

An axis can have a *label*, which is a textual representation of the attribute according to which the axis varies the position of the page. You can set the attribute using the following key:

`/tikz/data visualization/axis options/label={[`⟨*options*⟩`]`⟨*text*⟩`}` (default axis's label in math mode)

This key sets the label of an axis to ⟨*text*⟩. This text will typically be placed inside a `node` and the ⟨*options*⟩ can be used to further configure the way this node is rendered. The ⟨*options*⟩ will be executed with the path prefix `/tikz/data visualization/`, so you need to say `node style` to configure the styling of a node, see Section 82.4.7.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
    scientific axes,
    x axis = {label, length=2.5cm},
    y axis = {label={[node style={fill=blue!20}]{$x^2$}}},
    visualize as smooth line]
 data [format=function] {
   var x : interval [-3:5];
   func y = \value x * \value x;
 };
```

Note that using the `label` key does not actually cause a node to be created, because it is somewhat unclear where the label should be placed. Instead, the `visualize label` key is used (typically internally by an axis system) to show the label at some sensible position. This key is documented in Section 82.5.5.

82.2.8 Reference: Axis Types

As explained earlier, when you use `new axis base` to create a new axis, a powerful scaling and attribute mapping mechanism is installed, but no mapping of values to positions on the page is performed. For this, a *transformation object* must be installed. The following keys take care of this for you. Note, however, that even these keys do not cause a visual representation of the axis to be added to the visualization – this is the job of an axis system, see Section 82.3.

`/tikz/data visualization/new Cartesian axis=`⟨*name*⟩ (no default)

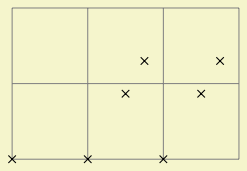
This key creates a new "Cartesian" axis, named ⟨*name*⟩. For such an axis, the (scaled) values of the axis's attribute are transformed into a displacement on the page along a straight line. The following key is used to configure in which "direction" the axis points:

`/tikz/data visualization/axis options/unit vector=`⟨*coordinate*⟩ (no default, initially `(1pt,0pt)`)

Recall that an axis takes the values of an attribute and rescales them so that they fit into a "reasonable" interval $[t_1, t_2]$. Suppose that v' is the rescaled dimension in (TEX) points. Then when the data point is visualized, the coordinate system will be shifted by v' times the ⟨*coordinate*⟩.

As an example, suppose that you have said `scaling=0 and 10pt and 50 and 20pt`. Then when the underlying attribute has the value `25`, it will be mapped to a v' of 15 (because `25` lies in the middle of `0` and `50` and `15pt` lies in the middle of `10pt` and `20pt`). This, in turn, causes the data point to be displaced by 15 times the ⟨*coordinate*⟩.

The bottom line is that the ⟨*coordinate*⟩ should usually denote a point that is at distance `1pt` from the origin and that points into the direction of the axis.



```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);

  \datavisualization
    [new Cartesian axis=x axis, x axis={attribute=x},
     new Cartesian axis=y axis, y axis={attribute=y},
     x axis={unit vector=(0:1pt)},
     y axis={unit vector=(60:1pt)},
     visualize as scatter]
  data {
    x, y
    0, 0
    1, 0
    2, 0
    1, 1
    2, 1
    1, 1.5
    2, 1.5
  };
\end{tikzpicture}
```

82.3 Axis Systems

An *axis system* is, as the name suggests, a whole family of axes that act in concert. For example, in the "standard" axis system there is a horizontal axis called the x-axis that monitors the `x` attribute (by default, you can change this easily) and a vertical axis called the y-axis. Furthermore, a certain number of ticks are added and labels are placed at sensible positions.

82.3.1 Usage

Using an axis system is usually pretty easy: You just specify a key like `scientific axes` and the necessary axes get initialized with sensible default values. You can then start to modify these default values, if necessary.

First, you can (and should) set the attributes to which the difference axes refer. For instance, if the `time` attribute is plotted along the x-axis, you would write

```
x axis = {attribute = time}
```

Second, you may wish to modify the lengths of the axes. For this, you can use keys like `length` or further keys as described in the references later on.

Third, you may often wish to modify how many ticks and grid lines are shown. By default, no grid lines are shown, but you can say the following in order to cause grid lines to be shown:

```
all axes={grid}
```

Naturally, instead of `all axes` you can also specify a single axis, causing only grid lines to be shown for this axis. In order to change the number of ticks that are shown, you can say

```
all axes={ticks=few}
```

or also `many` instead of `few` or even `none`. Far more fine-grained control over the tick placement and rendering is possible, see Section 82.4 for details.

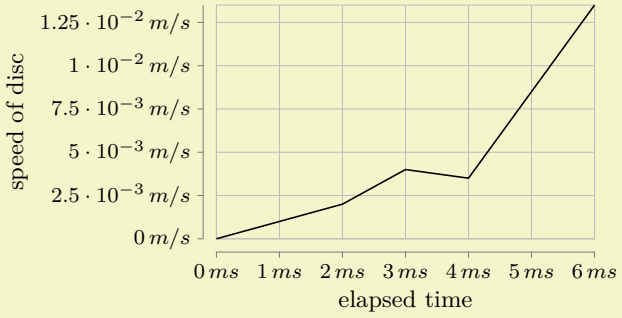
Fourth, consider adding units (like "cm" for centimeters or "m/s^2" for acceleration) to your ticks:

```
x axis={ticks={tick unit=cm}}, y axis={ticks={tick unit=m/s^2}}
```

Finally, consider adding labels to your axes. For this, use the label option:

```
x axes={time $t$ (ms)}, y axis={distance $d$ (mm)}
```

Here is an example that employs most of the above features:

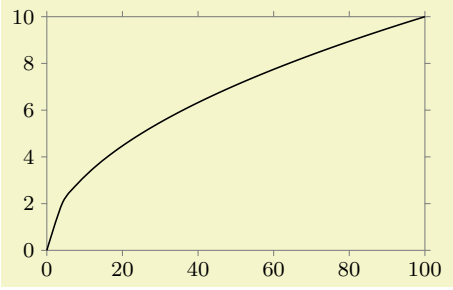



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  scientific axes=clean,
  x axis={attribute=time, ticks={tick unit=ms},
    label={elapsed time}},
  y axis={attribute=v, ticks={tick unit=m/s},
    label={speed of disc}},
  all axes=grid,
  visualize as line]
data {
  time, v
  0, 0
  1, 0.001
  2, 0.002
  3, 0.004
  4, 0.0035
  5, 0.0085
  6, 0.0135
};
```

82.3.2 Reference: Scientific Axis Systems

`/tikz/data visualization/scientific axes=`⟨*options*⟩ (no default)

This key installs a two-dimensional coordinate system based on the attributes `/data point/x` and `/data point/y`.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [scientific axes,
                      visualize as smooth line]
    data [format=function] {
      var x : interval [0:100];
      func y = sqrt(\value x);
    };
\end{tikzpicture}
```

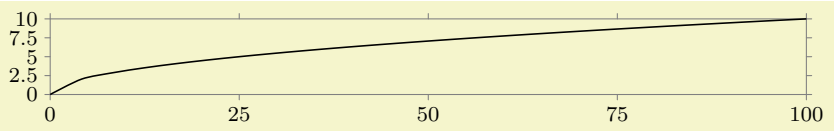
This axis system is usually a good choice to depict "arbitrary two dimensional data". Because the axes are automatically scaled, you do not need to worry about how large or small the values will be. The name `scientific axes` is intended to indicate that this axis system is often used in scientific publications.

You can use the ⟨*options*⟩ to fine tune the axis system. The ⟨*options*⟩ will be executed with the following path prefix:

```
/tikz/data visualization/scientific axes
```

All keys with this prefix can thus be passed as ⟨*options*⟩.

This axis system will always distort the relative magnitudes of the units on the two axis. If you wish the units on both axes to be equal, consider directly specifying the unit length "by hand":




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [visualize as smooth line,
                      scientific axes,
                      all axes={unit length=1cm per 10 units, ticks={few}}]
    data [format=function] {
      var x : interval [0:100];
      func y = sqrt(\value x);
    };
\end{tikzpicture}
```

The `scientific axes` have the following properties:

- The `x`-values are surveyed and the x-axis is then scaled and shifted so that it has the length specified by the following key.

  `/tikz/data visualization/scientific axes/width`=⟨*dimension*⟩ (no default, initially `5cm`)

  The minimum value is at the left end of the axis and at the canvas origin. The maximum value is at the right end of the axis.
- The `y`-values are surveyed and the y-axis is then scaled so that is has the length specified by the following key.

  `/tikz/data visualization/scientific axes/height`=⟨*dimension*⟩ (no default)

  By default, the `height` is the golden ratio times the `width`.

  The minimum value is at the bottom of the axis and at the canvas origin. The maximum value is at the top of the axis.
- Lines (forming a frame) are depicted at the minimum and maximum values of the axes in 50% black.

The following keys are executed by default as options: `outer ticks` and `standard labels`.

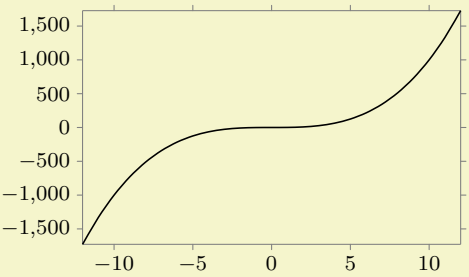
You can use the following style to overrule the defaults:

`/tikz/data visualization/every scientific axes` (style, no value)

The keys described in the following can be used to fine-tune the way the scientific axis system is rendered.

`/tikz/data visualization/scientific axes/outer ticks` (no value)

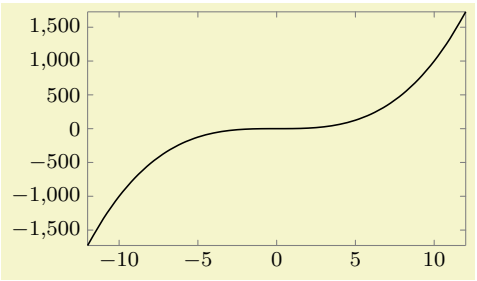
This causes the ticks to be drawn " on the outside" of the frame so that they interfere as little as possible with the data. It is the default.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [scientific axes=outer ticks,
                      visualize as smooth line]
    data [format=function] {
      var x : interval [-12:12];
      func y = \value x*\value x*\value x;
    };
\end{tikzpicture}
```

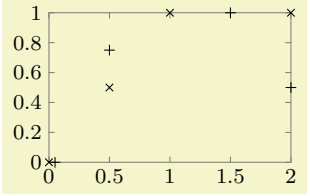
/tikz/data visualization/scientific axes/inner ticks (no value)

This axis system works like `scientific axes`, only the ticks are on the “inside” of the frame.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [scientific axes=inner ticks,
                      visualize as smooth line]
    data [format=function] {
      var x : interval [-12:12];
      func y = \value x*\value x*\value x;
    };
\end{tikzpicture}
```

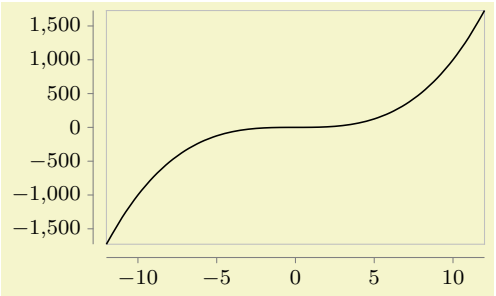
This axis system is also common in publications, but the ticks tend to interfere with marks if they are near to the border as can be seen in the following example:




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization [scientific axes={inner ticks, width=3.2cm},
                      style sheet=cross marks,
                      visualize as scatter/.list={a,b}]
    data [set=a] {
      x, y
      0, 0
      1, 1
      0.5, 0.5
      2, 1
    }
    data [set=b] {
      x, y
      0.05, 0
      1.5, 1
      0.5, 0.75
      2, 0.5
    };
\end{tikzpicture}
```

/tikz/data visualization/scientific axes/clean (no value)

The axes and the ticks are completely removed from the actual data, making this axis system especially useful for scatter plots, but also for most other scientific plots.



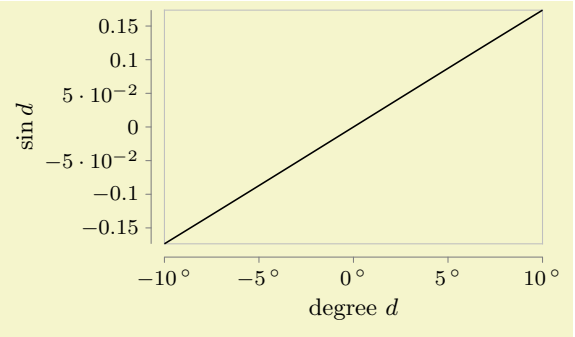

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line]
data [format=function] {
  var x : interval [-12:12];
  func y = \value x*\value x*\value x;
};
```

The distance of the axes from the actual plot is given by the padding of the axes.

For all scientific axis systems, different label placement strategies can be specified. They are discussed in the following.

/tikz/data visualization/scientific axes/standard labels (no value)

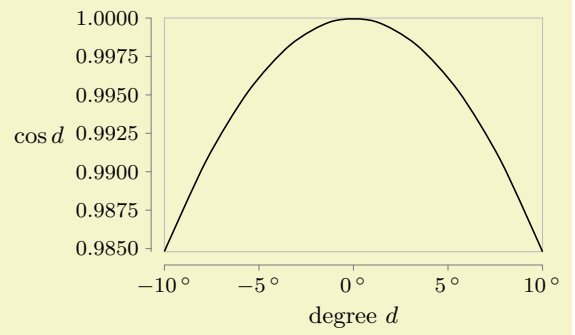
As the name suggests, this is the standard placement strategy. The label of the x-axis is placed below the center of the x-axis, the label of the y-axis is rotated by 90° and placed left of the center of the y-axis.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean, standard labels,
  visualize as smooth line,
  x axis={label=degree $d$,
    ticks={tick unit={}^\circ}},
  y axis={label=$\sin d$}]
data [format=function] {
  var x : interval [-10:10] samples 10;
  func y = sin(\value x);
};
```

/tikz/data visualization/scientific axes/upright labels (no value)

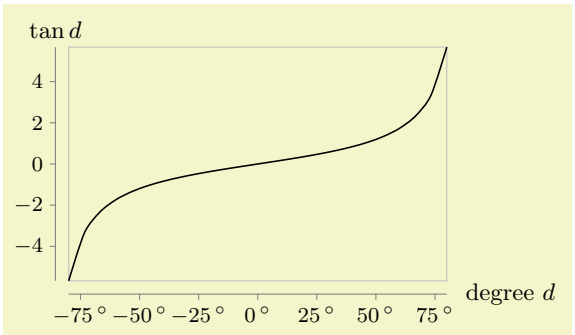
Works like `scientific axes standard labels`, only the label of the y-axis is not rotated.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean, upright labels,
  visualize as smooth line,
  x axis={label=degree $d$,
    ticks={tick unit={}^\circ}},
  y axis={label=$\cos d$, include value=1,
    ticks={style={
        /pgf/number format/precision=4,
        /pgf/number format/fixed zerofill}}}]
data [format=function] {
  var x : interval [-10:10] samples 10;
  func y = cos(\value x);
};
```

/tikz/data visualization/scientific axes/end labels (no value)

Places the labels at the end of the x- and the y-axis, similar to the axis labels of a school book axis system.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean, end labels,
  visualize as smooth line,
  x axis={label=degree $d$,
    ticks={tick unit={}^\circ}},
  y axis={label=$\tan d$}]
data [format=function] {
  var x : interval [-80:80];
  func y = tan(\value x);
};
```

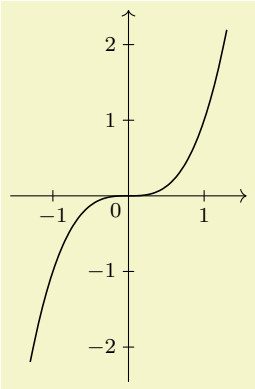
82.3.3 Reference: School Book Axis Systems

/tikz/data visualization/school book axes=⟨*options*⟩ (no default)

This axis system is intended to "look like" the coordinate systems often used in school books: The axes are drawn in such a way that they intersect to origin. Furthermore, no automatic scaling is done to ensure that the lengths of units are the same in all directions.

This axis system must be used with care – it is nearly always necessary to specify the desired unit length by hand using the option `unit length`. If the magnitudes of the units on the two axes differ, different unit lengths typically need to be specified for the different axes.

Finally, if the data is "far removed" from the origin, this axis system will also "look bad".



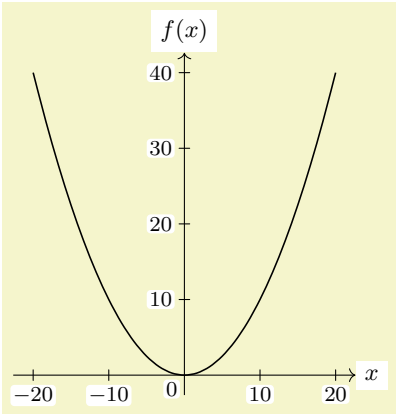

```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line]
    data [format=function] {
      var x : interval [-1.3:1.3];
      func y = \value x*\value x*\value x;
    };
\end{tikzpicture}
```

The stepping of the ticks is one unit by default. Using keys like `ticks=some` may help to give better steppings.

The ⟨*options*⟩ are executed with the key itself as path prefix. Thus, the following subkeys are permissible options:

`/tikz/data visualization/school book axes/unit`=⟨*value*⟩ (no default)

Sets the scaling so that 1 cm corresponds to ⟨*value*⟩ units. At the same time, the stepping of the ticks will also be set to ⟨*value*⟩.

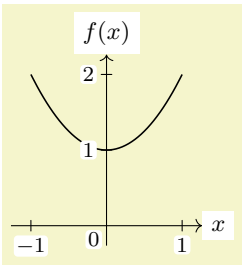



```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [school book axes={unit=10},
                      visualize as smooth line,
                      clean ticks,
                      x axis={label=$x$},
                      y axis={label=$f(x)$}]
    data [format=function] {
      var x : interval [-20:20];
      func y = \value x*\value x/10;
    };
\end{tikzpicture}
```

`/tikz/data visualization/school book axes/standard labels` (no value)

This key makes the label of the x-axis appear at the right end of this axis and it makes the label of the y-axis appear at the top of the y-axis.

Currently, this is the only supported placement strategy for the school book axis system.



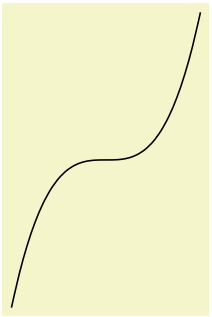

```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [school book axes={standard labels},
                      visualize as smooth line,
                      clean ticks,
                      x axis={label=$x$},
                      y axis={label=$f(x)$}]
    data [format=function] {
      var x : interval [-1:1];
      func y = \value x*\value x + 1;
    };
\end{tikzpicture}
```

82.3.4 Advanced Reference: Underlying Cartesian Axis Systems

The axis systems described in the following are typically not used directly by the user. The systems setup *directions* for several axes in some sensible way, but they do not actually draw anything on these axes. For instance, the `xy Cartesian` creates two axes called `x axis` and `y axis` and makes the x-axis point right and the y-axis point up. In contrast, an axis system like `scientific axes` uses the axis system `xy Cartesian` internally and then proceeds to setup a lot of keys so that the axis lines are drawn, ticks and grid lines are drawn, and labels are placed at the correct positions.

`/tikz/data visualization/xy Cartesian` (no value)

This axis system creates two axes called `x axis` and `y axis` that point right and up, respectively. By default, one unit is mapped to one cm.



```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [xy Cartesian, visualize as smooth line]
    data [format=function] {
      var x : interval [-1.25:1.25];
      func y = \value x*\value x*\value x;
    };
\end{tikzpicture}
```

`/tikz/data visualization/xy axes=`⟨*options*⟩ (no default)

This key applies the ⟨*options*⟩ both to the `x axis` and the `y axis`.

`/tikz/data visualization/xyz Cartesian cabinet` (no value)

This axis system works like `xy Cartesian`, only it *additionally* creates an axis called `z axis` that points left and down. For this axis, one unit corresponds to $\frac{1}{2}\sin 45°$cm. This is also known as a cabinet projection.

`/tikz/data visualization/xyz axes=`⟨*options*⟩ (no default)

This key applies the ⟨*options*⟩ both to the `x axis` and the `y axis`.

`/tikz/data visualization/uv Cartesian` (no value)

This axis system works like `xy Cartesian`, but it introduces two axes called `u axis` and `v axis` rather than the `x axis` and the `y axis`. The idea is that in addition to a "major" xy-coordinate system this is also a "smaller" or "minor" coordinate system in use for depicting, say, small vectors with respect to this second coordinate system.

`/tikz/data visualization/uv axes=`⟨*options*⟩ (no default)

Applies the ⟨*options*⟩ to both the `u axis` and the `y axis`.

`/tikz/data visualization/uvw Cartesian cabinet` (no value)

Like `xyz Cartesian cabinet`, but for the uvw-system.

`/tikz/data visualization/uvw axes=`⟨*options*⟩ (no default)

Like `xyz axes`.

82.4 Ticks and Grids

82.4.1 Concepts

A *tick* is a small visual indication on an axis of the value of the axis's attribute at the position where the tick is shown. A tick may be accompanied additionally by a textual representation, but it need not. A *grid line* is similar to a tick, but it is not an indication on the axis, but rather a whole line that indicates all positions where the attribute has a certain value. Unlike ticks, grid lines (currently) are not accompanied by a textual representation.

Just as for axes, the data visualization system decouples the specification of which ticks are present *in principle* from where they are visualized. In the following, I describe how you specify which ticks and grid lines you would like to be drawn and how they should look like (their styling). The axis system of your choice will then visualize the ticks at a sensible position for the chosen system. For details on how to change where whole axis is shown along with its ticks, see Section 82.5.4.

Specifying which ticks you are interested in is done as follows: First, you use `ticks` key (or, for specifying which grid lines should be present, the `grid` key). This key takes several possible options, described in detail in the following, which have different effects:

1. Keys like `step=10` or `minor steps between steps` cause a "semi-automatic" computation of possible steps. Here, you explicitly specify the stepping of steps, but the first stepping and their number are computed automatically according to the range of possible values for the attribute.

2. Keys like `few`, `some`, or `many` can be passed to `ticks` in order to have Ti*k*Z compute good tick positions automatically. This is usually what you want to happen, which is why most axis system will implicitly say `ticks={some}`.

3. Keys like `at` or `also at` provide "absolute control" over which ticks or grid lines are shown. For these keys, you can not only specify at what value a tick should be shown, but also its styling and also whether it is a major, minor, or subminor tick or grid line.

In the following, the main keys `ticks` and `grids` are documented first. Then the different kinds of ways of specifying where ticks or grid lines should be shown are explained.

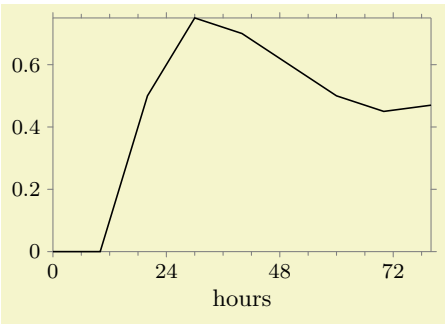
82.4.2 The Main Options: Tick and Grid

`/tikz/data visualization/axis options/ticks=`⟨*options*⟩ (default `some`)

This key can be passed to an axis in order to configure which ticks are present for the axis. The possible ⟨*options*⟩ include, for instance, keys like `step`, which is used to specify a stepping for the ticks, but also keys like `major` or `minor` for specifying the positions of major and minor ticks in detail. The list of possible options is described in the rest of this section.

Note that the `ticks` option will only configure which ticks should be shown in principle. The actual rendering is done only when the `visualize ticks` key is used, documented in Section 82.5.4, which is typically done only internally by an axis system.

The ⟨*options*⟩ will be executed with the path prefix `/tikz/data visualization/`. When the `ticks` key is used multiple times for an axis, the ⟨*options*⟩ accumulate.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  scientific axes, visualize as line,
  x axis={ticks={step=24, minor steps between steps=3},
          label=hours}]
  data {
    x, y
    0, 0
    10, 0
    20, 0.5
    30, 0.75
    40, 0.7
    50, 0.6
    60, 0.5
    70, 0.45
    80, 0.47
  };
```

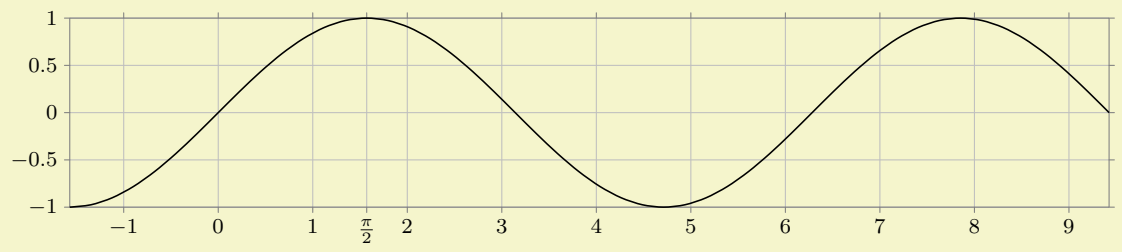
/tikz/data visualization/axis options/grid=⟨*options*⟩ (default `at default ticks`)

This key is similar to `ticks`, only it is used to configure where grid lines should be shown rather than ticks. In particular, the options that can be passed to the `ticks` key can also be passed to the `grid` key. Just like `ticks`, the ⟨*options*⟩ only specify which grid lines should be drawn in principle; it is the job of the `visualize grid` key to actually cause any grid lines to be shown.

If you do not specify any ⟨*options*⟩, the default text `at default ticks` is used. This option causes grid lines to be drawn at all positions where ticks are shown by default. Since this usually exactly what you would like to happen, most of the time you just need to `all axes=grid` to cause a grid to be shown.

/tikz/data visualization/axis options/ticks and grid=⟨*options*⟩ (no default)

This key passes the ⟨*options*⟩ to both the `ticks` key and also to the `grid` key. This is useful when you want to specify some special points explicitly where you wish a tick to be shown and also a grid line.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   visualize as smooth line,
   all axes= {grid, unit length=1.25cm},
   y axis={ ticks=few },
   x axis={ ticks=many, ticks and grid={ major also at={(pi/2) as $\frac{\pi}{2}$}}}]
  data [format=function] {
    var x : interval [-pi/2:3*pi] samples 50;
    func y = sin(\value x r);
  };
```

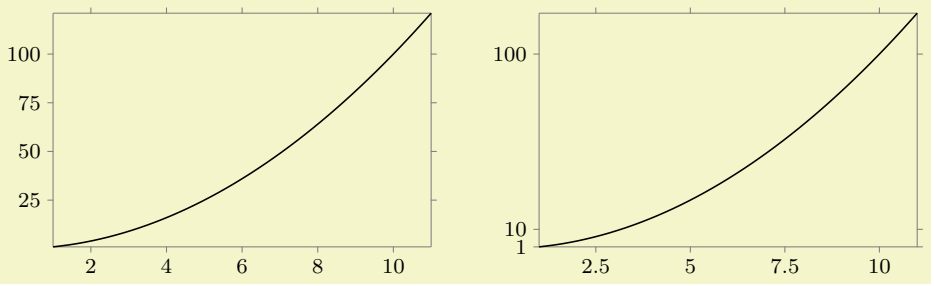
82.4.3 Semi-Automatic Computation of Tick and Grid Line Positions

Consider the following problem: The data visualization engine determines that in a plot the x-values vary between 17.4 and 34.5. In this case, we certainly do not want, say, ten ticks at exactly ten evenly spaced positions starting with 17.4 and ending with 34.5, because this would yield ticks at positions like 32.6. Ticks should be placed at "nice" positions like 20, 25, and 30.

Determining which positions are "nice" is somewhat difficult. In the above example, the positions 20, 25, and 30 are certainly nice, but only three ticks may be a bit few of them. Better might be the tick positions 17.5, 20, 22.5, through to 32.5. However, users might prefer even numbers over fractions like 2.5 as the stepping.

A *tick placement strategy* is a method of automatically deciding which positions are *good* for placing ticks. The data visualization engine comes with a number of predefined strategies, but you can also define new ones yourself. When the data visualization is requested to automatically determine "good" positions for the placement of ticks on an axis, it uses one of several possible *basic strategies*. These strategies differ dramatically in which tick positions they will choose: For a range of values between 5 and 1000, a `linear steps` strategy might place ticks at positions 100, 200, through to 1000, while an `exponential steps` strategy would prefer the tick positions 10, 100 and 1000. The exact number and values of the tick positions chosen by either strategy can be fine-tuned using additional options like `step` or `about`.

Here is an example of the different stepping chosen when one varies the tick placement strategy:



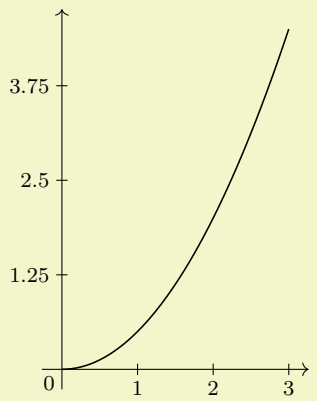

```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [scientific axes, visualize as smooth line]
    data [format=function] {
      var x : interval [1:11];
      func y = \value x*\value x;
    };
\end{tikzpicture}
\qquad
\begin{tikzpicture}
  \datavisualization [scientific axes, visualize as smooth line,
    y axis={exponential steps},
    x axis={ticks={quarter about strategy}},
  ]
    data [format=function] {
      var x : interval [1:11];
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

Two strategies are always available: `linear steps`, which yields (semi)automatic ticks are evenly spaced positions, and `exponential steps`, which yields (semi)automatic steps at positions at exponentially increasing positions – which is exactly what is needed for logarithmic plots. These strategies are details in Section 82.4.13.

The following options are used to configure tick placement strategies like `linear steps`. Unlike the basic choice of a placement strategy, which is an axis option, the following should be passed to the option `ticks` or `grid` only. So, you would write things like `x axis={ticks={step=2}}`, but `x axis={linear steps}`.

`/tikz/data visualization/step`=⟨*value*⟩ (no default, initially `1`)

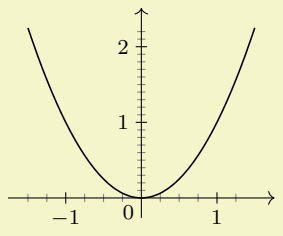
The value of this key is used to determine the spacing of the major ticks. The key is used by the `linear steps` and `exponential steps` strategies, see the explanations in Section 82.4.13 for details. Basically, all ticks are placed at all multiples of ⟨*value*⟩ that lie in the attribute range interval.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
    school book axes, visualize as smooth line,
    y axis={ticks={step=1.25}},
  ]
    data [format=function] {
      var x : interval [0:3];
      func y = \value x*\value x/2;
    };
```

`/tikz/data visualization/minor steps between steps`=⟨*number*⟩ (default `9`)

Specifies that between any two major steps (whose positions are specified by the `step` key), there should be ⟨*number*⟩ many minor steps. Note that the default of `9` is exactly the right number so that each interval between two minor steps is exactly a tenth of the size of a major step. See also Section 82.4.13 for further details.




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [school book axes, visualize as smooth line,
    x axis={ticks={minor steps between steps=3}},
    y axis={ticks={minor steps between steps}},
  ]
    data [format=function] {
      var x : interval [-1.5:1.5];
      func y = \value x*\value x;
    };
\end{tikzpicture}
```

/tikz/data visualization/phase=⟨*value*⟩ (no default, initially 0)

See Section 82.4.13 for details on how the phase of steps influences the tick placement.

82.4.4 Automatic Computation of Tick and Grid Line Positions

The `step` option gives you "total control" over the stepping of ticks on an axis, but you often do not know the correct stepping in advance. In this case, you may prefer to have a good value for `step` being computed for you automatically.

Like the `step` key, these options are passed to the `ticks` option. So, for instance, you would write `x axis={ticks={about=4}}` to request about four ticks to be placed on the x-axis.

/tikz/data visualization/about=⟨*number*⟩ (no default)

This key asks the data visualization to place *about* ⟨*number*⟩ many ticks on an axis. It is not guaranteed that *exactly* ⟨*number*⟩ many ticks will be used, rather the actual number will be the closest number of ticks to ⟨*number*⟩ so that their stepping is still "good". For instance, when you say `about=10`, it may happen that exactly `10`, but perhaps even `13` ticks are actually selected, provided that these numbers of ticks lead to good stepping values like `5` or `2.5` rather than numbers like `3.4` or `7`. The method that is used to determine which steppings a deemed to be "good" depends on the current tick placement strategy.

Linear steps. Let us start with `linear steps`: First, the difference between the maximum value $v_{\max}$ and the minimum value $v_{\min}$ on the axis is computed; let us call it r for "range". Then, r is divided by ⟨*number*⟩, yielding a target stepping s. If s is a number like 1 or 5 or 10, then this number could be used directly as the new value of `step`. However, s will typically something strange like 0.023 45 or 345 223.76, so s must be replaced by a better value like 0.02 in the first case and perhaps 250 000 in the second case.

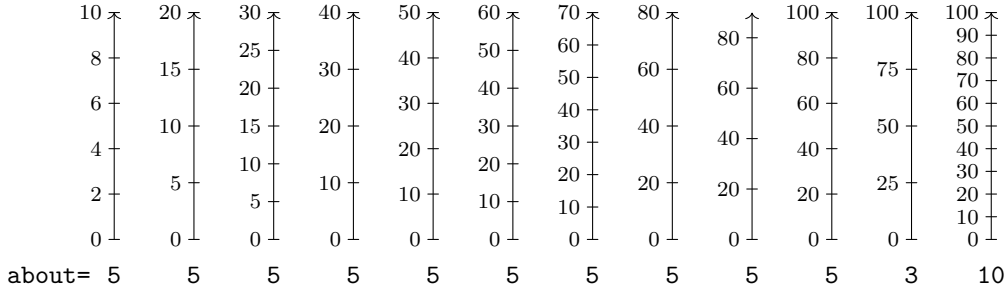
In order to determine which number is to be used, s is rewritten in the form $m \cdot 10^k$ with $1 \leq m < 10$ and $k \in \mathbb{Z}$. For instance, 0.023 45 would be rewritten as $2.345 \cdot 10^{-2}$ and 345 223.76 as $3.452\,2376 \cdot 10^5$. The next step is to replace the still not-so-good number m like 2.345 or 3.452 237 by a "good" value m'. For this, the current value of the `about strategy` is used:

/tikz/data visualization/about strategy=⟨*list*⟩ (no default)

The ⟨*list*⟩ is a comma-separated sequence of pairs ⟨*threshold*⟩/⟨*value*⟩ like for instance `1.5/1.0` or `2.3/2.0`. When a good value m' is sought for a given m, we iterate over the list and find the first pair ⟨*threshold*⟩/⟨*value*⟩ where ⟨*threshold*⟩ exceeds m. Then m' is set to ⟨*value*⟩. For instance, if ⟨*list*⟩ is `1.5/1.0,2.3/2.0,4/2.5,7/5,11/10`, which is the default, then for $m = 3.141$ we would get $m' = 2.5$ since $4 > 3.141$, but $2.3 \leq 3.141$. For $m = 6.3$ we would get $m' = 5$.

Once m' has been determined, the stepping is set to $s' = m' \cdot 10^k$.

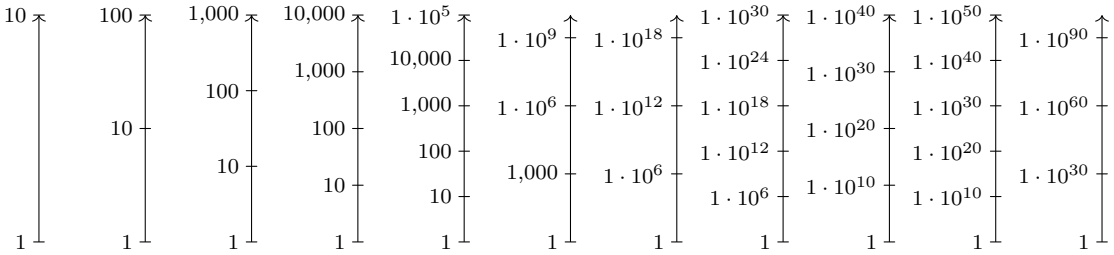
The net effect of all this is that for the default strategy the only valid stepping are the values 1, 2, 2.5 and 5 and every value obtainable by multiplying one of these values by a power of ten. The following example shows the effects of, first, setting `about=5` (corresponding to the `some` option) and then having axes where the minimum value is always `0` and where the maximum value ranges from `10` to `100` and, second, setting `about` to the values from `3` (corresponding to the `few` option) and to `10` (corresponding to the `many` option) while having the minimum at `0` and the maximum at `100`:



Exponential steps. For `exponential steps` the strategy for determining a good stepping value is similar to `linear steps`, but with the following differences:

- Naturally, since the stepping value refers to the exponent, the whole computation of a good stepping value needs to be done "in the exponent". Mathematically spoken, instead of considering the difference $r = v_{\max} - v_{\min}$, we consider the difference $r = \log v_{\max} - \log v_{\min}$. With this difference, we still compute $s = r/\langle number \rangle$ and let $s = m \cdot 10^k$ with $1 \le m < 10$.
- It makes no longer sense to use values like 2.5 for m' since this would yield a fractional exponent. Indeed, the only sensible values for m' seem to be 1, 3, 6, and 10. Because of this, the `about strategy` is ignored and one of these values or a multiple of one of them by a power of ten is used.

The following example shows the chosen steppings for a maximum varying from 10^1 to 10^5 and from 10^{10} to 10^{50} as well as for 10^{100} for `about=3`:



Alternative strategies.

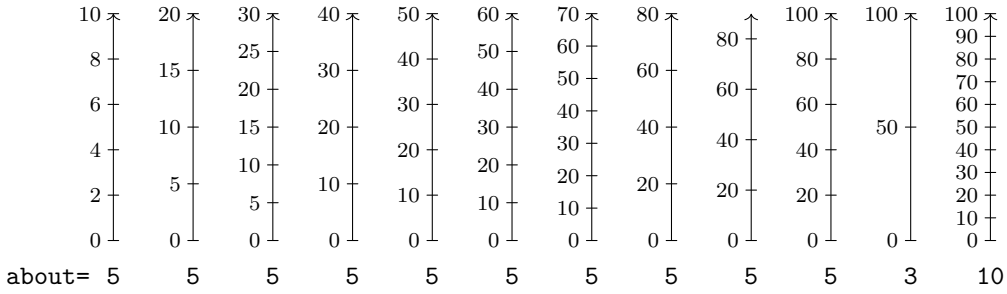
In addition to the standard `about strategy`, there are some additional strategies that you might wish to use instead:

`/tikz/data visualization/standard about strategy` (no value)

Permissible values for m' are: 1, 2, 2.5, and 5. This strategy is the default strategy.

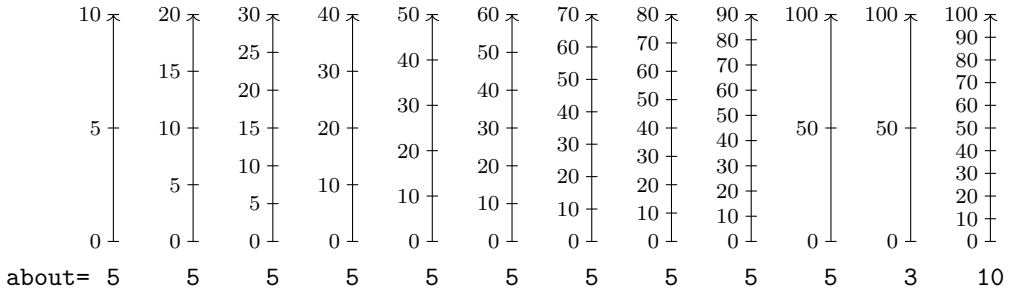
`/tikz/data visualization/euro about strategy` (no value)

Permissible values for m' are: 1, 2, and 5. These are the same values as for the Euro coins, hence the name.



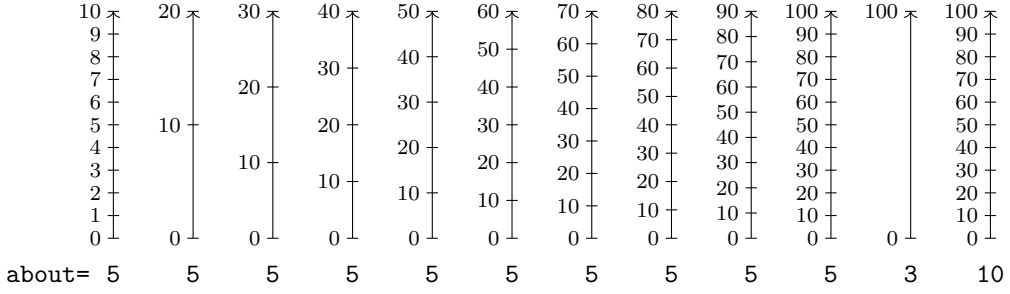
`/tikz/data visualization/half about strategy` (no value)

Permissible values for m': 1 and 5. Use this strategy if only powers of 10 or halves thereof seem logical.



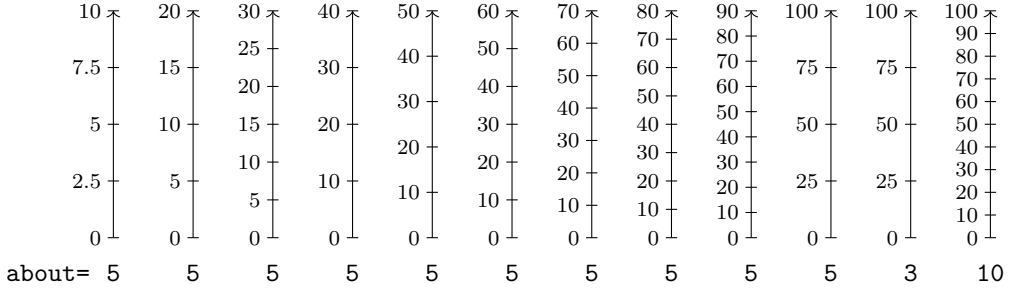
/tikz/data visualization/decimal about strategy (no value)

The only permissible value for m' is 1. This is an even more radical version of the previous strategy.



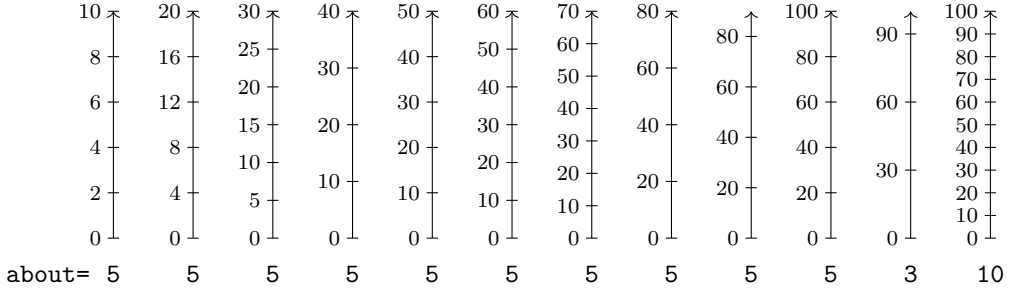
/tikz/data visualization/quarter about strategy (no value)

Permissible values for m' are: 1, 2.5, and 5.



/tikz/data visualization/int about strategy (no value)

Permissible values for m' are: 1, 2, 3, 4, and 5.



/tikz/data visualization/many (no value)

This is an abbreviation for `about=10`.

/tikz/data visualization/some (no value)

This is an abbreviation for `about=5`.

/tikz/data visualization/few (no value)

This is an abbreviation for `about=3`.

/tikz/data visualization/none (no value)

Switches off the automatic step computation. Unless you use `step=` explicitly to set a stepping, no ticks will be (automatically) added.

82.4.5 Manual Specification of Tick and Grid Line Positions

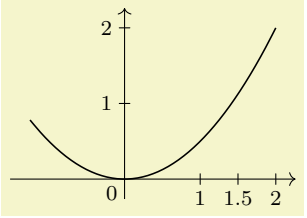
The automatic computation of ticks and grid lines will usually do a good job, but not always. For instance, you might wish to have ticks exactly at, say, prime numbers or at Fibonacci numbers or you might wish to have an additional tick at π. In these cases you need more direct control over the specification of tick positions.

First, it is important to understand that the data visualization system differentiates between three kinds of ticks and grid lines: major, minor, and subminor. The major ticks are the most prominent ticks where, typically, a textual representation of the tick is shown; and the major grid lines are the thickest. The minor ticks are smaller, more numerous, and lie between major ticks. They are used, for instance, to indicate positions in the middle between major ticks or at all integer positions between major ticks. Finally, subminor ticks are even smaller than minor ticks and they lie between minor ticks.

Four keys are used to configure the different kinds:

/tikz/data visualization/major=⟨*options*⟩ (no default)

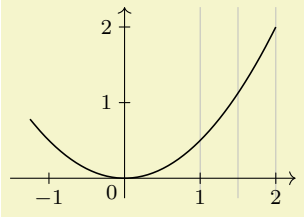
The key can be passed as an option to the `ticks` key and also to the `grid` key, which in turn is passed as an option to an axis. The ⟨*options*⟩ passed to `major` specify at which positions major ticks/grid lines should be shown (using the `at` option and `also at` option) and also any special styling. The different possible options are described later in this section.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={ticks={major={at={1, 1.5, 2}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

/tikz/data visualization/minor=⟨*options*⟩ (no default)

Like `major`, only for minor ticks/grid lines.



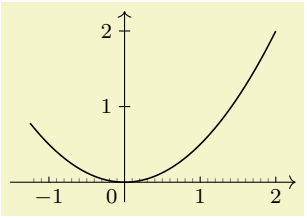

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={grid={minor={at={1, 1.5, 2}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

/tikz/data visualization/subminor=⟨*options*⟩ (no default)

Like `major`, only for subminor ticks/grid lines.

/tikz/data visualization/common=⟨*options*⟩ (no default)

This key allows you to specify ⟨*options*⟩ that apply to `major`, `minor` and `subminor` alike. It does not make sense to use `common` to specify positions (since you typically do not want both a major and a minor tick at the same position), but it can be useful to configure, say, the size of all kinds of ticks:



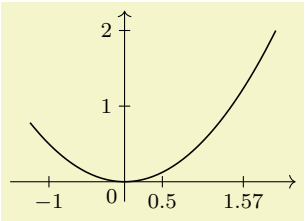

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={ticks={minor steps between steps, common={low=0}}} ]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

The following keys can now be passed to the `major`, `minor`, and `subminor` keys to specify where ticks or grid lines should be shown:

`/tikz/data visualization/at=`⟨*list*⟩ (no default)

Basically, the ⟨*list*⟩ must be a list of values that is processed with the `\foreach` macro (thus, it can contain ellipses to specify ranges of value). Empty values are skipped.

The effect of passing `at` to a `major`, `minor`, or `subminor` key is that ticks or grid lines on the axis will be placed exactly at the values in ⟨*list*⟩. Here is an example:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={ticks={major={at={-1,0.5,(pi/2)}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

When this option is used, any previously specified tick positions are overwritten by the values in ⟨*list*⟩. Automatically computed ticks are also overwritten. Thus, this option gives you complete control over where ticks should be placed.

Normally, the individual values inside the ⟨*list*⟩ are just numbers that are specified in the same way as an attribute value. However, such a value may also contain the keyword `as`, which allows you so specify the styling of the tick in detail. Section 82.4.6 details how this works.

It is often a bit cumbersome that one has to write things like

```
some axis = {ticks = {major = {at = {...}}}}
```

A slight simplification is given by the following keys, which can be passed directly to `ticks` and `grid`:

`/tikz/data visualization/major at=`⟨*list*⟩ (no default)

A shorthand for `major={at={`⟨*list*⟩`}}`.

`/tikz/data visualization/minor at=`⟨*list*⟩ (no default)

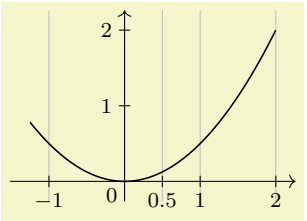
A shorthand for `major={at={`⟨*list*⟩`}}`.

`/tikz/data visualization/subminor at=`⟨*list*⟩ (no default)

A shorthand for `major={at={`⟨*list*⟩`}}`.

`/tikz/data visualization/also at=`⟨*list*⟩ (no default)

This key is similar to `at`, but it causes ticks or grid lines to be placed at the positions in the ⟨*list*⟩ *in addition* to the ticks that have already been specified either directly using `at` or indirectly using keys like `step` or `some`. The effect of multiple calls of this key accumulate. However, when `at` is used after an `also at` key, the `at` key completely resets the positions where ticks or grid lines are shown.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [ school book axes, visualize as smooth line,
    x axis={grid, ticks and grid={major={also at={0.5}}}}]
  data [format=function] {
    var x : interval [-1.25:2];
    func y = \value x * \value x / 2;
  };
```

As for `at`, there are some shorthands available:

`/tikz/data visualization/major also at=`⟨*list*⟩ (no default)

A shorthand for `major={also at={`⟨*list*⟩`}}`.

`/tikz/data visualization/minor also at=`⟨*list*⟩ (no default)

A shorthand for `major={also at={`⟨*list*⟩`}}`.

`/tikz/data visualization/subminor also at=`⟨*list*⟩ (no default)

A shorthand for `major={also at={`⟨*list*⟩`}}`.

82.4.6 Styling Ticks and Grid Lines: Introduction

When a tick, a tick label, or a grid line is visualized on the page, a whole regiment of styles influences the appearance. The reason for this large number of interdependent styles is the fact that we often wish to influence only a very certain part of how a tick is rendered while leaving the other aspects untouched: Sometimes we need to modify just the font of the tick label; sometimes we wish to change the length of the tick label and the tick label position at the same time; sometimes we wish to change the color of grid line, tick, and tick label; and sometimes we wish to generally change the thickness of all ticks.

Let us go over the different kinds of things that can be styled (grid lines, ticks, and tick labels) one by one and let us have a look at which styles are involved. We will start with the grid lines, since they turn out to be the most simple, but first let us have a look at the general `style` and `styling` mechanism that is used in many placed in the following:

82.4.7 Styling Ticks and Grid Lines: The Style and Node Style Keys

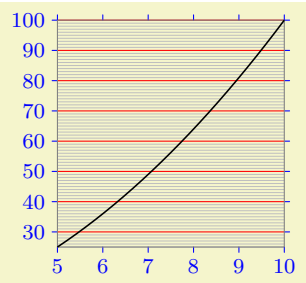
All keys of the data visualization system have the path prefix `/tikz/data visualization`. This is not only true for the main keys like `scientific axes` or `visualize as line`, but also for keys that govern how ticks are visualized. In particular, a style like `every major grid` has the path prefix `/tikz/data visualization` and all keys stored in this style are also executed with this path prefix.

Normally, this does not cause any trouble since most of the keys and even styles used in a data visualization are intended to configure what is shown in the visualization. However, at some point, we may also with to specify options that no longer configure the visualization in general, but specify the appearance of a line or a node on the Ti*k*Z layer.

Two keys are used to "communicate" with the Ti*k*Z layer:

`/tikz/data visualization/style=`⟨*TikZ options*⟩ (no default)

This key takes options whose path prefix is `/tikz`, not `/tikz/data visualization`. These options will be *appended* to a current list of such options (thus, multiple calls of this key accumulate). The resulting list of keys is not executed immediately, but it will be executed whenever the data visualization engine calls the Ti*k*Z layer to draw something (this placed will be indicated in the following).




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   all axes={ticks={style=blue}, length=3cm},
   y axis={grid, grid={minor steps between steps, major={style=red}}},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

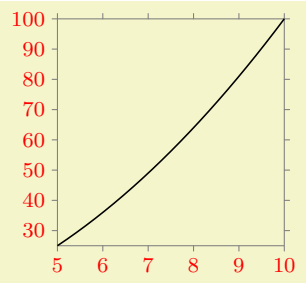
`/tikz/data visualization/styling` (no value)

Executing this key will cause all "accumulated" Ti*k*Z options from previous calls to the key `/tikz/data visualization/style` to be executed. Thus, you use `style` to set Ti*k*Z options, but you use `styling` to actually apply these options. Usually, you do not call this option directly since this application is only done deep inside the data visualization engine.

Similar to `style` (and `styling`) there also exist the `node style` (and `node styling`) key that takes Ti*k*Z options that apply to nodes only – in addition to the usual `style`.

`/tikz/data visualization/node style=`⟨*TikZ options*⟩ (no default)

This key works like `style`, but it has an effect only on nodes that are created during a data visualization. This includes tick labels and axis labels:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   all axes={ticks={node style=red}, length=3cm},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

Note that in the example the ticks themselves (the little thicker lines) are not red.

`/tikz/data visualization/node styling` (no value)

Executing this key will cause all "accumulated" node stylings to be executed.

82.4.8 Styling Ticks and Grid Lines: Styling Grid Lines

When a grid line is visualized, see Section 82.5.3 for details on when this happens, the following styles are executed in the specified order.

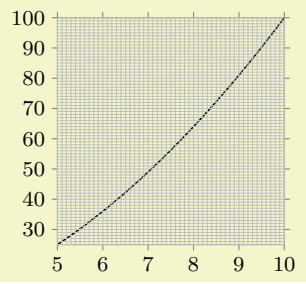
1. `grid layer`.
2. `every grid`.
3. `every major grid` or `every minor grid` or `every subminor grid`, depending on the kind of grid line.
4. locally specified options for the individual grid line, see Section 82.4.11.
5. `styling`, see Section 82.4.7.

All of these keys have the path prefix `/tikz/data visualization`. However, the options stored in the first style (`grid layer`) and also in the last (`styling`) are executed with the path prefix `/tikz` (see Section 82.4.7).

Let us now have a look at these keys in detail:

`/tikz/data visualization/grid layer` (style, initially `on background layer`)

This key is used to specified the *layer* on which grid lines should be drawn (layers are explained in Section 45). By default, all grid lines are placed on the `background` layer and thus behind the data visualization. This is a sensible strategy since it avoids obscuring the more important data with the far less important grid lines. However, you can change this style to "get the grid lines to the front":




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   all axes={
     length=3cm,
     grid,
     grid={minor steps between steps}
   },
   grid layer/.style=, % none, so on top of data (bad idea)
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

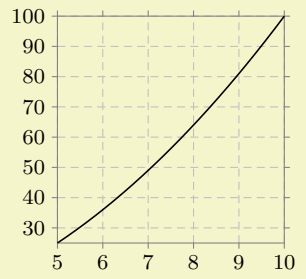
When this style is executed, the keys stored in the style will be executed with the prefix `/tikz`. Normally, you should only set this style to be empty or to `on background layer`.

`/tikz/data visualization/every grid` (style, no value)

This style provides overall configuration options for grid lines. By default, it is set to the following:

```
low=min, high=max
```

This causes grid lines to span all possible values when they are visualized, which is usually the desired behavior (the `low` and `high` keys are explained in Section 82.5.4. You can append the `style` key to this style to configure the overall appearance of grid lines. It should be noted that settings to `style` inside `every grid` will take precedence over ones in `every major grid` and `every minor grid`. In the following example we cause all grid lines to be dashed (which is not a good idea in general since it creates a distracting background pattern).



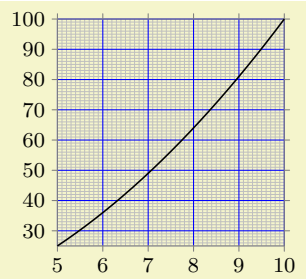

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   all axes={length=3cm, grid},
   every grid/.append style={style=densely dashed},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

`/tikz/data visualization/every major grid` (style, no value)

This style configures the appearance of major grid lines. It does so by calling the `style` key to setup appropriate Ti*k*Z options for visualizing major grid lines. The default definition of this style is:

```
style = {help lines, thin, black!25}
```

In the following example, we use thin major blue grid lines:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   all axes={
     length=3cm,
     grid,
     grid={minor steps between steps}
   },
   every major grid/.style = {style={blue, thin}},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

As can be seen, this is not exactly visually pleasing. The default settings for the grid lines should work in most situations; you may wish to increase the blackness level, however, when you experience trouble during printing or projecting graphics.

`/tikz/data visualization/every minor grid` (style, no value)

Works like `every major grid`. The default is

```
style = {help lines, black!25}
```

`/tikz/data visualization/every subminor grid` (style, no value)

Works like `every major grid`. The default is

```
style = {help lines, black!10}
```

82.4.9 Styling Ticks and Grid Lines: Styling Ticks and Tick Labels

Styling ticks and tick labels is somewhat similar to styling grid lines. Let us start with the tick *mark*, that is, the small line that represents the tick. When this mark is drawn, the following styles are applied:

1. `every ticks`.

2. `every major ticks` or `every minor ticks` or `every subminor ticks`, depending on the kind of ticks to be visualized.
3. locally specified options for the individual tick, see Section 82.4.11.
4. `tick layer`
5. `every odd tick` or `every even tick`, see Section 82.4.12.
6. `draw`
7. `styling`, see Section 82.4.7.

For the tick label node (the node containing the textual representation of the attribute's value at the tick position), the following styles are applied:

1. `every ticks`.
2. `every major ticks` or `every minor ticks` or `every subminor ticks`, depending on the kind of ticks to be visualized.
3. locally specified options for the individual tick, see Section 82.4.11.
4. `tick node layer`
5. `every odd tick` or `every even tick`, see Section 82.4.12.
6. `styling`, see Section 82.4.7.
7. `node styling`, see Section 82.4.7.

`/tikz/data visualization/every ticks` (style, no value)

This style allows you to configure the appearance of ticks using the `style` and `node style` key. Here is (roughly) the default definition of this style:

```
node style={
  font=\footnotesize,
  inner sep=1pt,
  outer sep=.1666em,
  rounded corners=1.5pt
}
```

`/tikz/data visualization/every major ticks` (style, no value)

The default is

```
  style={line cap=round}, tick length=2pt
```

`/tikz/data visualization/every minor ticks` (style, no value)

The default is

```
  style={help lines,thin, line cap=round}, tick length=1.4pt
```

`/tikz/data visualization/every subminor ticks` (style, no value)

The default is

```
  style={help lines, line cap=round}, tick length=0.8pt
```

`/tikz/data visualization/tick layer` (style, initially `on background layer`)

Like `grid layer`, this key specifies on which layer the ticks should be placed.

`/tikz/data visualization/tick node layer` (style, initially empty)

Like `tick layer`, but now for the nodes. By default, tick nodes are placed on the main layer and thus on top of the data in case that the tick nodes are inside the data.

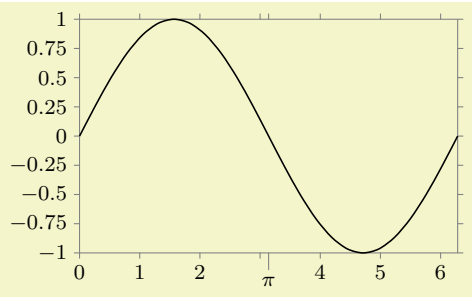
82.4.10 Styling Ticks and Grid Lines: Exceptional Ticks

You may sometimes wish to style a few ticks differently from the other ticks. For instance, in the axis system `school book axes` there should be a tick label at the `0` position only on one axis and then this label should be offset a bit. In many cases this is easy to achieve: When you add a tick "by hand" using the `at` or `also at` option, you can add any special options in square brackets.

However, in some situations the special tick position has been computed automatically for you, for instance by the `step` key or by saying `tick=some`. In this case, adding a tick mark with the desired options using `also at` would cause the tick mark with the correct options to be shown in addition to the tick mark with the wrong options. In cases like this one, the following option may be helpful:

`/tikz/data visualization/options at=`⟨*value*⟩ `as [`⟨*options*⟩`]` (no default)

This key causes the ⟨*options*⟩ to be executed for any tick mark(s) at ⟨*value*⟩ in addition to any options given already for this position:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line,
  x axis={ticks={major={
    options at = 3    as [no tick text],
    also at    = (pi) as
      [{tick text padding=1ex}] $\pi$}}}]
data [format=function] {
  var x : interval[0:2*pi];
  func y = sin(\value x r);
};
```

`/tikz/data visualization/no tick text at=`⟨*value*⟩ (no default)

Shorthand for `options at=`⟨*value*⟩ `as [no tick text]`.

82.4.11 Styling Ticks and Grid Lines: Styling and Typesetting a Value

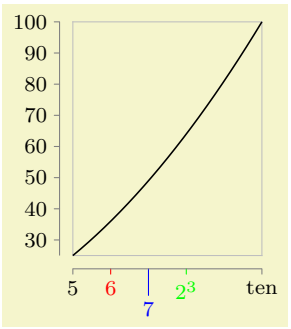
The `at` and `also at` key allow you to provide a comma-separated ⟨*list*⟩ of ⟨*value*⟩s where ticks or grid lines should be placed. In the simplest case, the ⟨*value*⟩ is simply a number. However, the general syntax allows three different kinds of ⟨*value*⟩s:

1. ⟨*value*⟩
2. ⟨*value*⟩ `as [`⟨*local options*⟩`]`
3. ⟨*value*⟩ `as [`⟨*local options*⟩`]` ⟨*text*⟩

In the first case, the ⟨*value*⟩ is just a number that is interpreted like any other attribute value.

In the second case, where the keyword `as` is present, followed by some option in square brackets, but nothing following the closing square bracket, when the tick or grid line at position ⟨*value*⟩ is shown, the ⟨*local options*⟩ are executed first. These can use the `style` key or the `node style` key to configure the appearance of this single tick or grid line. You can also use keys like `low` or `high` to influence how large the grid lines or the ticks are or keys like `tick text at low` to explicitly hide or show a tick label.

In the third case, which is only important for `ticks` and not for `grid`, the same happens as in the second case, but the text that is shown as tick label is ⟨*text*⟩ rather than the automatically generated tick label. This automatic generation of tick labels is explained in the following.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes=clean,
   x axis={length=2.5cm, ticks={major at={
         5,
         6 as [style=red],
         7 as [{style=blue, low=-1em}],
         8 as [style=green] $2^3$,
         10 as ten
       }}},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

A value like "2" or "17" could just be used as ⟨*text*⟩ to be displayed in the node of a tick label. However, things are more difficult when the to-be-shown value is 0.0000000015, because we then would typically (but not always) prefer something like $1.5 \cdot 10^{-9}$ to be shown. Also, we might wish a unit to be added like 23m/s. Finally, we might wish a number like 3.141 to be replaced by π. For these reasons, the data visualization system does not simply put the to-be-shown value in a node as plain text. Instead, the number is passed to a *typesetter* whose job it is to typeset this number nicely using TEX's typesetting capabilities. The only exception is, as indicated above, the third syntax version of the `at` and `also at` keys, where ⟨*text*⟩ is placed in the tick label's node, regardless of what the typesetting would usually do.

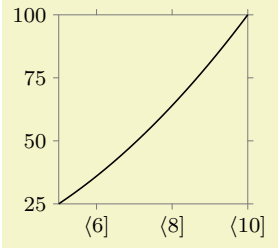
The text produced by the automatic typesetting is computed as follows:

1. The current contents of the key `tick prefix` is put into the node.
2. This is followed by a call of the key `tick typesetter` which gets the ⟨*value*⟩ of the tick as its argument in scientific notation.
3. This is followed by the contents of the key `tick suffix`.

Let us have a look at these keys in detail:

`/tikz/data visualization/tick prefix`=⟨*text*⟩ (no default, initially empty)

The ⟨*text*⟩ will be put in front of every typeset tick:



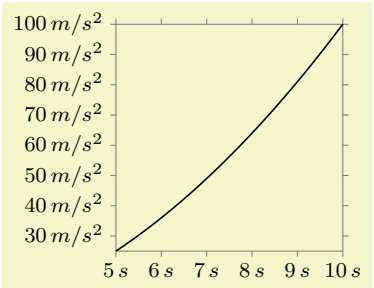

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes, all axes={ticks=few, length=2.5cm},
   x axis={ticks={tick prefix=$\langle$, tick suffix=$]$}},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

`/tikz/data visualization/tick suffix`=⟨*text*⟩ (no default, initially empty)

Works like `tick prefix`. This key is especially useful for adding units like "cm" or "m/s" to every tick label. For this reason, there is a (near) alias that is easier to memorize:

`/tikz/data visualization/tick unit`=⟨*roman math text*⟩ (no default)

A shorthand for `tick suffix={$\,\rm`⟨*roman math text*⟩`$}`:

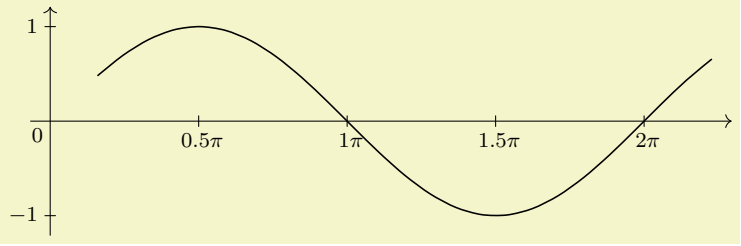


```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes, all axes={length=3cm},
   x axis={ticks={tick unit=s}},
   y axis={ticks={tick unit=m/s^2}},
   visualize as line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

/tikz/data visualization/tick typesetter=⟨*value*⟩ (no default)

The key gets called for each number that should be typeset. The argument ⟨*value*⟩ will be in scientific notation (like `1.0e1` for 10). By default, this key applies `\pgfmathprintnumber` to its argument. This command is a powerful number printer whose configuration is documented in Section 97.

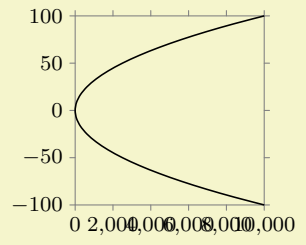
You are invited to code underlying this key so that a different typesetting mechanism is used. Here is a (not quite finished) example that shows how, say, numbers could be printed in terms of multiples of π:




```
\usetikzlibrary {datavisualization.formats.functions}
\def\mytypesetter#1{%
  \pgfmathparse{#1/pi}%
  \pgfmathprintnumber{\pgfmathresult}$\pi$%
}
\tikz \datavisualization
  [school book axes, all axes={unit length=1.25cm},
   x axis={ticks={step=(0.5*pi), tick typesetter/.code=\mytypesetter{##1}}},
   y axis={include value={-1,1}},
   visualize as smooth line]
  data [format=function] {
    var x : interval [0.5:7];
    func y = sin(\value x r);
  };
```

82.4.12 Stacked Ticks

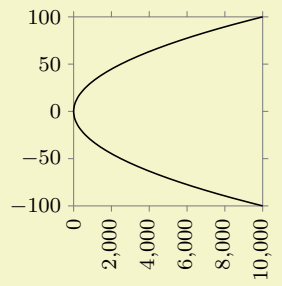
Sometimes, the text of tick labels are so long or so numerous that the text of adjacent tick labels overlap (or have too little padding):




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

There are two ways to address this problem:

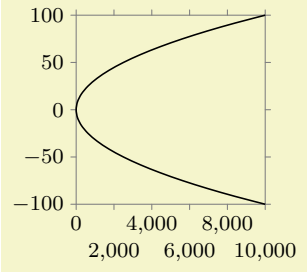
- One can rotate the labels on horizontal axes:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          x axis={ticks={node style={rotate=90, anchor=east}}},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

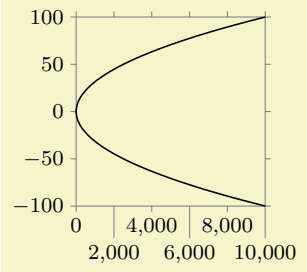
This is often a good solution, but may be hard to read. Also consider rotating labels only by 45° or 30°.

- One can specify different shifts of the nodes for the different ticks, whereby the ticks text no longer overlap.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          x axis={ticks={major at={0,4000,8000,
                                2000 as [node style={yshift=-1em}],
                                6000 as [node style={yshift=-1em}],
                                10000 as [node style={yshift=-1em}]}}},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

However, specifying shifts "by hand" in the above way is not always an option, especially when the tick positions should be computed automatically. Instead, the `stack` option can be used, which is much easier to use and gives better results:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm}, x axis={ticks=stack},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

The `stack` option is actually just a style that gives you access to the general even/odd mechanism for ticks with labels. Whenever a tick mark is created where a tick label is also to be drawn, two special things happen:

1. For every odd tick mark, the `every odd tick` style is executed, for every even tick mark the `every even tick`. Here, "odd" and "even" are with respect to the order in which the ticks have been added to the list of `at` positions for each major, minor, or subminor tick list, not with respect to the order in which they will appear on the axis. Thus, when you write

   ```
   ticks={major at={1,2,3,4}, major at={0,-1,-2}, minor at={9,8,7}}
   ```

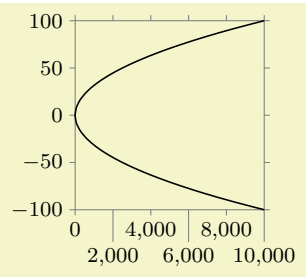
   then for `1`, `3`, `0`, and `-2` as well as `9` and `7` the key `every odd tick` will be executed, while `every even tick` will be executed for positions `2`, `4`, `-1`, and also `8`.

2. When a tick node label is shown at the `low` position of the tick mark, the dimension stored in the key `tick text low even padding` is added to the `low` value. Provided that this padding is not zero (which is the default), the length of the even tick marks will be increased and the tick label node will be placed at a greater distance from the axis.

   Similar keys exist for padding ticks with labels at high positions and also at even positions.

/tikz/data visualization/tick text low even padding=⟨*dimension*⟩ (no default, initially `0pt`)

When a tick label is shown at the low position of an even tick, the ⟨*distance*⟩ is added to the `low` value, see also Section 82.5.4.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          x axis={ticks={tick text low even padding=-1em}},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

Note that ⟨*dimension*⟩ should usually be non-positive.

The following keys work similarly:

/tikz/data visualization/tick text low odd padding=⟨*dimension*⟩ (no default, initially `0pt`)

/tikz/data visualization/tick text high even padding=⟨*dimension*⟩ (no default, initially `0pt`)

/tikz/data visualization/tick text high odd padding=⟨*dimension*⟩ (no default, initially `0pt`)

/tikz/data visualization/tick text odd padding=⟨*dimension*⟩ (no default)

A shorthand for setting `tick text odd low padding` and `tick text odd high padding` at the same time.

/tikz/data visualization/tick text even padding=⟨*dimension*⟩ (no default)

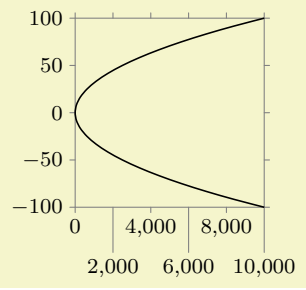
A shorthand for setting `tick text even low padding` and `tick text even high padding` at the same time.

/tikz/data visualization/tick text padding=⟨*dimension*⟩ (no default)

Sets all text paddings to ⟨*dimension*⟩.

/tikz/data visualization/stack=⟨*dimension*⟩ (default `1em`)

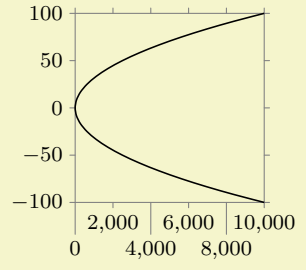
Shorthand for `tick text even padding`=⟨*dimension*⟩.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          x axis={ticks={stack=1.5em}},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

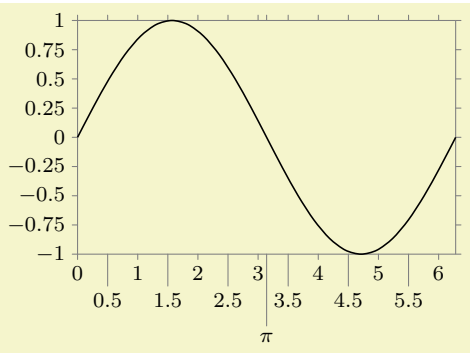
/tikz/data visualization/stack'=⟨*dimension*⟩ (no default)

Shorthand for `tick text odd padding`=⟨*dimension*⟩. The difference to `stack` is that the set of value that are "lowered" is exactly exchanged with the set of value "lowered" by `stack`.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes,
                          all axes={length=2.5cm},
                          x axis={ticks=stack'},
                          visualize as smooth line]
  data [format=function] {
    var y : interval[-100:100];
    func x = \value y*\value y;
  };
```

Note that the above keys have an effect on all tick labels of an axis, also on special ticks that you may have added using the `also at` key. When using the `stack` key, you should specify a `tick text padding` explicitly for such keys:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
  [scientific axes,
   x axis={ticks={stack, many, major also at=
     {(pi) as [{tick text padding=2.5em}] $\pi$}}},
   visualize as smooth line]
  data [format=function] {
    var x : interval[0:(2*pi)];
    func y = sin(\value x r);
  };
```

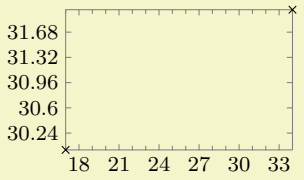
82.4.13 Reference: Basic Strategies

`/tikz/data visualization/axis options/linear steps` (no value)

This strategy places ticks at positions that are evenly spaced by the current value of `step`.

In detail, the following happens: Let a be the minimum value of the data values along the axis and let b be the maximum. Let the current *stepping* be s (the stepping is set using the `step` option, see below) and let the current *phasing* be p (set using the `phase`) option. Then ticks are placed all positions $i \cdot s + p$ that lie in the interval $[a, b]$, where i ranges over all integers.

The tick positions computed in the way described above are *major* step positions. In addition to these, if the key `minor steps between steps` is set to some number n, then n many minor ticks are introduced between each two major ticks (and also before and after the last major tick, provided the values still lie in the interval $[a, b]$). Note that is n is 1, then one minor tick will be added in the middle between any two major ticks. Use a value of 9 (not 10) to partition the interval between two major ticks into ten equally sized minor intervals.



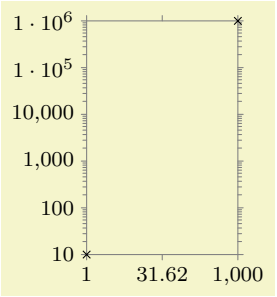

```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization
    [scientific axes={inner ticks, width=3cm},
     x axis={ticks={step=3, minor steps between steps=2}},
     y axis={ticks={step=.36}},
     visualize as scatter]
    data {
      x, y
      17, 30
      34, 32
    };
\end{tikzpicture}
```

`/tikz/data visualization/axis options/exponential steps` (no value)

This strategy produces ticks at positions that are appropriate for logarithmic plots. It is automatically selected when you use the `logarithmic` option with an axis.

In detail, the following happens: As for `linear steps` let numbers a, b, s, and p be given. Then, major ticks are placed at all positions $10^{i \cdot s + p}$ that lie in the interval $[a, b]$ for $i \in \mathbb{Z}$.

The minor steps are added in the same way as for `linear steps`. In particular, they interpolate *linearly* between major steps.




```
\usetikzlibrary {datavisualization}
\begin{tikzpicture}
  \datavisualization
    [scientific axes,
     x axis={logarithmic, length=2cm, ticks={step=1.5}},
     y axis={logarithmic, ticks={step=1, minor steps between steps=9}},
     visualize as scatter]
    data {
      x, y
      1, 10
      1000, 1000000
    };
\end{tikzpicture}
```

82.4.14 Advanced: Defining New Placement Strategies

`/tikz/data visualization/axis options/tick placement strategy`=⟨*macro*⟩ (no default)

This key can be used to install a so-called *tick placement strategy*. Whenever `visualize ticks` is used to request some ticks to be visualized, it is checked whether some automatic ticks should be created. This is the case when the following key is set:

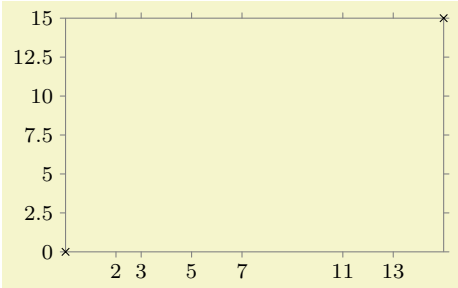
`/tikz/data visualization/compute step`=⟨*code*⟩ (no default)

The ⟨*code*⟩ should compute a suitable value for the stepping to be used by the ⟨*macro*⟩ in the tick placement strategy.

For instance, the `step` key sets `compute step` to `\def\tikz@lib@dv@step{#1}`. Thus, when you say `step=5`, then the desired stepping of `5` is communicated to the ⟨*macro*⟩ via the macro `\tikz@lib@dv@step`.

Provided `compute step` is set to some nonempty value, upon visualization of ticks the ⟨*macro*⟩ is executed. Typically, ⟨*macro*⟩ will first call the ⟨*code*⟩ stored in the key `compute step`. Then, it should implement some strategy then uses the value of the computed or desired stepping to create appropriate `at` commands. To be precise, it should set the keys `major`, `minor`, and/or `subminor` with some appropriate `at` values.

Inside the call of ⟨*macro*⟩, the macro `\tikzdvaxis` will have been set to the name of the axis for which default ticks need to be computed. This allows you to access the minimum and the maximum value stored in the `scaling mapper` of that axis.




```
\usetikzlibrary {datavisualization}
\def\silly{
  \tikzdatavisualizationset{major={at={
        2,3,5,7,11,13}}}
}
\begin{tikzpicture}
  \datavisualization [
    scientific axes, visualize as scatter,
    x axis={tick placement strategy=\silly}
    ]
    data {
      x, y
      0, 0
      15, 15
    };
\end{tikzpicture}
```

82.5 Advanced: Creating New Axis Systems

The `datavisualization` library comes with a number of predefined axis systems, like `scientific axes=clean`, but it is also possible and to define new axis systems. Doing so involves the following steps:

1. Creating a number of axes.
2. Configuring attributes of these axes like their length or default scaling.
3. Creating visual representations of the axes.

4. Creating visual representations of the ticks and grid lines.

The first step uses `new ... axis` keys to create new axes, the last steps use `visualize ...` keys to create the visual representations of the axes.

Note that the axis system has no control over the actual attribute value ranges and neither over which ticks need to be drawn. The axis system can only provide good defaults and then specify *how* the ticks or labels should be drawn and *where* on the page – but not at which values.

In the following, as a running example let us develop an axis system `our system` that does the following: For the x-axis is looks like a normal scientific axis system, but there are actually two y-axes: One at the left and one at the right, each using a different attribute, but both coexisting in the same picture.

82.5.1 Creating the Axes

A new axis system is created as a style key with the prefix `/tikz/data visualization`. Thus, we would write:

```
\tikzset{
  data visualization/our system/.style={
    ...
  }
}
```

In our system we need three axis: The x-axis, the left axis and the right axis. Since all of these axes are Cartesian axes, we write the following:

```
\tikzset{
  data visualization/our system/.style={
    new Cartesian axis=x axis,
    new Cartesian axis=left axis,
    new Cartesian axis=right axis,
    x axis={attribute=x},
    left axis={unit vector={(0cm,1pt)}},
    right axis={unit vector={(0cm,1pt)}},
  }
}
```

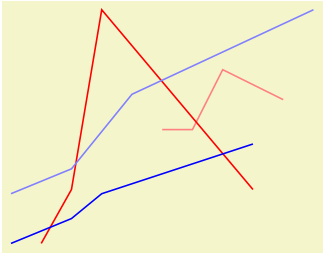
As can be seen, we also configure things so that the x-axis will use the `x` attribute by default (users can later change this by saying `x axis={attribute=`⟨*some other attribute*⟩`}`), but we do not configure the attributes of the `left axis` nor the `right axis`. We also make the left and right axis point upward (the `x axis` needs no configuration here since a Cartesian axis points right by default). The reason is the `left` would not be a particularly good attribute name and this way we ensure that users have to pick names themselves (hopefully good ones).

The next step is to define a standard scaling for the axes. Here, we can use the same as for `scientific axes`, so we would add the following keys to the definition of `our system`:

```
x axis    ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}}
```

We now already have enough to try our system, although we will not yet see any axes or ticks, but we will see the correct scaling of the attributes. Let us first define a data group:

```
\tikz \datavisualization data group {people and money} = {
    data [set=people 1] {
      time, people
      1900, 1000000000
      1920, 1500000000
      1930, 2000000000
      1980, 3000000000
    }
    data [set=people 2] {
      time, people
      1900, 2000000000
      1920, 2500000000
      1940, 4000000000
      2000, 5700000000
    }
    data [set=money 1] {
      time, money
      1910, 1.1
      1920, 2
      1930, 5
      1980, 2
    }
    data [set=money 2] {
      time, money
      1950, 3
      1960, 3
      1970, 4
      1990, 3.5
    }
  };
```



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2},
    people 1={style={visualizer color=blue}},
    people 2={style={visualizer color=blue!50}},
    money 1={style={visualizer color=red}},
    money 2={style={visualizer color=red!50}}]
  data group {people and money};
```

82.5.2 Visualizing the Axes

We must now show the axes themselves. For this we can use the `visualize axis` key:

`/tikz/data visualization/axis options/visualize axis`=⟨*options*⟩ (no default)

This key is passed to an axis as an option. It causes a visual representation of the axis to be created during the data visualization. The ⟨*options*⟩ are used to determine where the axis should be drawn and how long it should be. We can specify, for instance, that an axis should be drawn at the minimum value of another axis or where another axis has the value `0`.

The goto, high, and low Keys. In our example, the `left axis` should be shown at the left hand side. This is the position where the `x axis` has its minimum value. To specify this, we would use the following code:

```
left axis={ visualize axis={ x axis={ goto=min } }
```

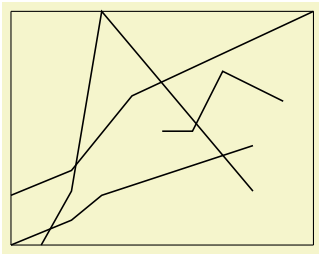
As can be seen, we can pass another axis as an ⟨*option*⟩ to `visualize axis`, where we pass the following key to the axis in turn:

`/tikz/data visualization/axis options/goto`=⟨*value*⟩ (no default)

The key can be passed to an axis. It will set the attribute monitored by the axis to the given ⟨*value*⟩, which is usually some number. However, ⟨*value*⟩ may also be one of the following, which causes a special behavior:

- `min`: The attribute is set to the minimal value that the attribute has attained along this axis.
- `max`: Like `min`.
- `padded min`: This will also set the ⟨*attribute*⟩ monitored by the axis to the same value as `min`. Additionally, however, the subkey `/data point/`⟨*attribute*⟩`/offset` is set to the current padding for the minimum, see the description of `padding min` later on. The effect of this is that the actual point "meant" by the attribute is offset by this padding along the attribute's axis.
- `padded max`: Like `padded min`.

The `right axis` would be visualized the same way, only at `goto=max`. The x-axis actually needs to be visualized *twice*: Once at the bottom and once at the top. Thus, we need to call `visualize axis` twice for this axis:



```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    left axis= {visualize axis={x axis=   {goto=min}}},
    right axis={visualize axis={x axis=   {goto=max}}},
    x axis=    {visualize axis={left axis={goto=min}},
                visualize axis={left axis={goto=max}}},
 }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

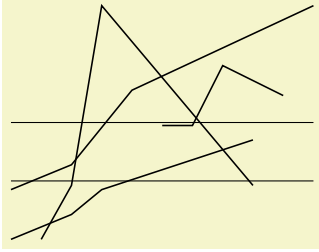
There is another key that is similar to `goto`, but has a slightly different semantics:

`/tikz/data visualization/axis options/goto pos=`⟨*fraction*⟩ (no default)

The key works like `goto`, only the ⟨*fraction*⟩ is not interpreted as a value but as a fraction of the way between the minimum and the maximum value for this axis.

Suppose that for an axis the attribute range interval is $[500, 1000]$ and the reasonable interval is $[1, 3]$. Then for a ⟨*fraction*⟩ of `0`, the mapping process would choose value 1 from the reasonable interval, for a ⟨*fraction*⟩ of `1` the position 3 from the reasonable interval, and for a ⟨*fraction*⟩ or `0.25` the position 1.5 since it is one quarter at the distance from 1 to 3.

Note that neither the attribute range interval nor the transformation function for the attribute are important for the `goto pos` option – the ⟨*fraction*⟩ is computed with respect to the reasonable interval. Also note that the values of the actual attribute corresponding to the fractional positions in the reasonable interval are not computed.



```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    x axis=    {visualize axis={left axis={goto pos=0.25}},
                visualize axis={left axis={goto pos=0.5}}},
 }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

By default, when an axis is visualized, it spans the set of all possible values for the monitored attribute, that is, from `min` to `max`. However, there are actually two keys that allow you to adjust this:

`/tikz/data visualization/low=`⟨*value*⟩ (no default)

This is the attribute value where the axis visualization starts. The same special values as for `goto` are permissible (like `min` or `padded min`, but also `0` or `1`).

/tikz/data visualization/high=⟨*value*⟩ (no default)

Like `low`, only for where the axis ends.

By default, `low=min` and `high=max` are set for an axis visualization. Another sensible setting is `low=padded min` and `high=padded max`. The following key provides a shorthand for this:

/tikz/data visualization/padded (no value)

Shorthand for `low=padded min, high=padded max`.

As an example, consider the `scientific axes=clean`. Here, each axis is actually drawn three times: Once at the minimum, once at the maximum and then once more at the padded minimum.

The axis line. When an axis is drawn, Ti*k*Z does not simply draw a straight line from the `low` position to the `high` position. In reality, the data visualization system uses the two commands `\pgfpathdvmoveto` and `\pgfpathdvlineto` internally. These will replace the straight line by a curve in certain situations. For instance, in a polar coordinate system, if an axis should be drawn along an angle axis for a fixed radius, an arc will be used instead of a straight line.

Styling the axis. As can be seen, we now get the axis we want (but without the ticks, visualizing them will be explained later). The axis is, however, simply a black line. We can *style* the axis in a manner similar to styling ticks and grid lines, see Section 82.4.7. In detail, the following styles get executed:

1. `axis layer`
2. `every axis`
3. `styling`

Additionally, even before `every axis` is executed, `low=min` and `high=max` are executed.

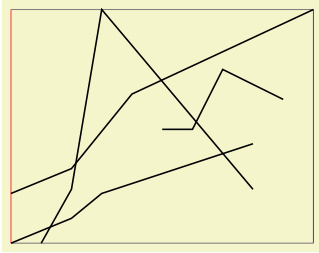
/tikz/data visualization/axis layer (style, initially `on background layer`)

The layer on which the axis is drawn. See the description of `grid layer` on page 897 for details.

/tikz/data visualization/every axis (style, no value)

Put styling of the axis here. It is usually a good idea to set this style to `style={black!50}`.

Recall that the `styling` key is set using the `style` key, see Section 82.4.7.



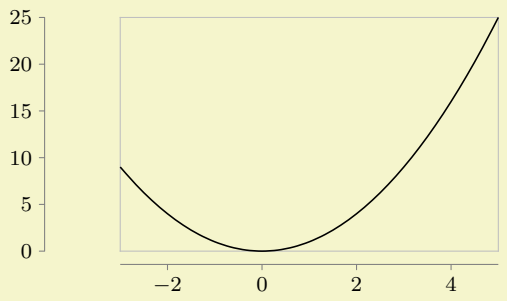
```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    every axis/.style={style=black!50}, % make this the default
    left axis= {visualize axis={x axis=   {goto=min}, style=red!75}},
    right axis={visualize axis={x axis=   {goto=max}, style=blue!75}},
    x axis=    {visualize axis={left axis={goto=min}},
                visualize axis={left axis={goto=max}}},
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=4cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

Padding the Axis. When an axis is visualized, it is often a good idea to make it "a little bit longer" or to "remove it a bit from the border", because the visualization of an axis should not interfere with the actual data. For this reason, a *padding* can be specified for axes:

/tikz/data visualization/axis options/padding min=⟨*dimension*⟩ (no default)

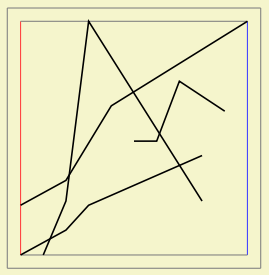
This is the dimension that is used whenever `goto=padded min` is used. The ⟨*dimension*⟩ is then put into the `offset` subkey of the attribute monitored by the axis. When a data point is transformed by a linear transformer and when this subkey is nonzero, this offset is added. (For an angle axis of a polar transformer, the ⟨*dimension*⟩ is interpreted as an additional angle rather than as an additional

distance). Note that ⟨*dimension*⟩ should typically be negative since "adding the ⟨*dimension*⟩" will then make the axis longer (because it starts at a smaller value). The standard axis systems set the padding to some default and take its value into account:




```
\usetikzlibrary {datavisualization.formats.functions}
\begin{tikzpicture}
  \datavisualization [scientific axes=clean,
                      x axis={padding min=-1cm},
                      visualize as smooth line]
    data [format=function] {
      var x : interval [-3:5];
      func y = \value x * \value x;
    };
\end{tikzpicture}
```

Using padded and using the `padded` key, we can visualize our axis "a little removed from the actual data":



```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    all axes=  {padding=.5em},
    left axis= {visualize axis={x axis=   {goto=padded min}, padded}},
    right axis={visualize axis={x axis=   {goto=padded max}, padded}},
    x axis=    {visualize axis={left axis={goto=padded min}, padded},
                visualize axis={left axis={goto=padded max}, padded}},
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

`/tikz/data visualization/axis options/padding max`=⟨*dimension*⟩ (no default)

Works like `padding min`, but ⟨*dimension*⟩ should typically be positive.

`/tikz/data visualization/axis options/padding`=⟨*dimension*⟩ (no default)

Sets both `padding min` to the negated value of ⟨*dimension*⟩ and `padding max` to ⟨*dimension*⟩.

82.5.3 Visualizing Grid Lines

As explained earlier, the `grid` key is used to specify at which positions grid lines should be drawn in principle. However, this key does not actually cause any grid lines to be drawn. Instead, the `visualize grid` key is used by the axis system to specify how grid lines are drawn.

`/tikz/data visualization/axis options/visualize grid`=⟨*options*⟩ (no default)

This key is passed to an axis. It causes grid lines to be drawn at the positions specified by the `grid` key for this axis. The ⟨*options*⟩ govern where and how the grid lines will be drawn.

The direction axis. At first sight, one might expect that the grid lines for an axis should simply be drawn perpendicular to the axis between the minimum and maximum value of the axis. However, things are somewhat more difficult in reality:

1. A grid line is supposed to indicate all positions where a certain attribute attains a fixed value. But, then, a grid line does not really need to be a grid *line*. Consider for instance a three dimensional axis system. A "grid line" for the x-coordinate `3` would actually be a "grid plane".
2. For a polar coordinate system and a fixed radius, this set of positions at a certain radius is not a straight line, but an arc. For more complicated coordinate systems such as the one arising from three-dimensional spherical projections, a grid line may well be a fairly involved curve.

The `visualize grid` command addresses these complications as follows:

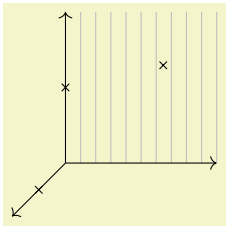
1. A grid line is always a line, not a plane or a volume. This means that in the example of a three dimensional axis system and the x-attribute being `3`, one would have to choose whether the grid line should go "along" the y-axis or "along" the z-axis for this position. One can, however, call the `visualize grid` command twice, once for each direction, to cause grid lines to be shown for both directions.
2. A grid line is created by moving to a start position and then doing a lineto to the target position. However, the "moveto" and "lineto" are done by calling special commands of the data visualization system. These special commands allow coordinate system to "notice" that the line is along an axis and will allow them to replace the straight line by an appropriate curve. The polar axes systems employ this strategy, for instance.

By the above discussion, in order to create a grid line for attribute a having value v, we need to specify an axis "along" which the line should be drawn. When there are only two axes, this is usually "the other axis". This "other axis" is specified using the following key:

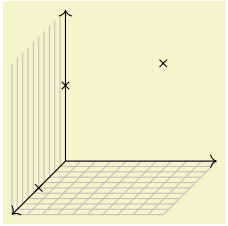
`/tikz/data visualization/direction axis=`⟨*axis name*⟩ (no default)

You must pass this key as an ⟨*option*⟩ each time you use `visualize axis`. When the grid line is drawn, the attribute a is set to v and the axis ⟨*axis name*⟩'s attribute is set once to the current value of `low` and once to `high`. Then a line is drawn between these two positions using `\pgfpathdvlineto`.

The `low` and `high` keys are the same as the ones used in the `visualize axis` key.



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
    xyz Cartesian cabinet,
    all axes={visualize axis={low=0, style=->}},
    x axis={visualize grid={direction axis=y axis}, grid=many},
    visualize as scatter]
  data {
    x, y, z
    0, 0, 1
    0, 1, 0
    2, 2, 2
  };
```

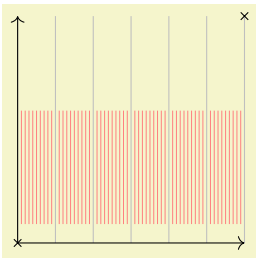


```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
    xyz Cartesian cabinet,
    all axes={visualize axis={low=0, style=->}, grid=many},
    x axis={visualize grid={direction axis=z axis}},
    z axis={visualize grid={direction axis=x axis},
            visualize grid={direction axis=y axis},},
    visualize as scatter]
  data {
    x, y, z
    0, 0, 1
    0, 1, 0
    2, 2, 2
  };
```

Styling the grid lines. When a grid line is draw, styles are applied as described in Section 82.4.8.

The major, minor, and subminor grid lines. The `grid` option allows you to specify for each kind of grid line (major, minor, or subminor) a set of different values for which these grid lines should be drawn. Correspondingly, it is also possible to configure for each kind of grid line how it should be drawn. For this, the `major`, `minor`, `subminor`, and also the `common` keys can be used inside the ⟨*options*⟩ of `visualize grid`. While as option to `grid` these keys are used to specify `at` values, as options of `visualize grid` they are used to configure the different kinds of grid lines.

Most of the time, no special configuration is necessary since all styling is best done by configuring keys like `every major grid`. You need to use a key like `major` only if you wish to configure for instance the `low` or `high` values of a `major` grid line differently from those of `minor` grid lines – are rather unlikely setting – or when the styling should deviate from the usual settings.



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
    xy Cartesian,
    all axes={visualize axis={low=0, style=->},
              grid={some, minor steps between steps}},
    x axis=  {visualize grid={
               direction axis=y axis,
               minor={low=0.25, high=1.75, style=red!50}}},
    visualize as scatter]
  data {
    x, y
    0, 0
    3, 3
  };
```

Returning to the example of `our system` with the two axis systems, it is straight-forward to configure the grid lines of the x-axis: The direction axis is either of the other two axis (they point in the same direction and they have the same range). For the other two axes, we visualize one grid independently of the other, using different colors.



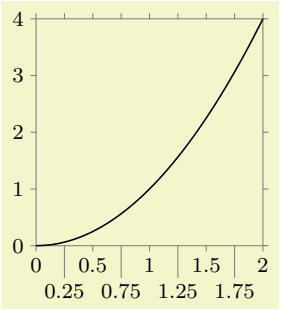
```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    x axis=    {visualize grid={direction axis=left axis}},
    left axis= {visualize grid={direction axis=x axis,
                                common={style=red!50}}},
    right axis={visualize grid={direction axis=x axis,
                                common={style=blue!50}}},
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, grid=many},
    left axis ={attribute=money, grid=some},
    right axis={attribute=people, grid=few},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

82.5.4 Visualizing the Ticks and Tick Labels

`/tikz/data visualization/axis options/visualize ticks=`⟨*options*⟩ (no default)

Visualizing a tick involves (possibly) drawing a tick mark and adding (possibly) the tick node. The process is similar to `visualize grid`: Users use the `ticks` key to configure how many ticks they would like for an axis and at which positions. The axis system uses the `visualize ticks` key to specify where these ticks should actually be shown.

Unlike grid lines, which are typically only visualized once for each combination of an axis and a direction axis, tick marks might be visualized at different places for the same axis. Consider for instance the `scientific axes`:

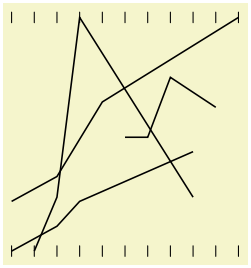



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes, all axes={length=3cm},
                          x axis={ticks={stack}},
                          visualize as smooth line]
  data [format=function] {
    var x : interval [0:2];
    func y = \value  x*\value x;
  };
```

Have a look at the ticks on the y-axis: There are ticks at values `0`, `1`, `2`, `3`, and `4`. These are visualized both at the left side (where the tick nodes are also shown) and additionally also at the right side, but only as small marks. Similarly, the ticks on the x-axis appear at the bottom, but also (in much simpler versions) at the top. Both for the x-axis and for the y-axis the `visualize ticks` key was called twice.

The tick marks. Drawing a tick mark is quite similar to visualizing a grid line; indeed a tick mark can be thought of as a "mini grid line": Just like a grid line it "points a long an axis". However, a tick will always be a short straight line – even when the coordinate system is actually twisted (experimentation has shown that ticks that follow the curvature of the coordinate system like grid lines are hard to recognize). For this reason, the `low` and `high` keys have a different meaning from the one used with the `visualize grid` key. In detail to configure the size and position of a tick mark for the value v of attribute a, proceed as follows:

- The `visualize ticks` key will have setup attribute a to be equal to v.
- You should now use the `goto` or `goto pos` key together with all *other* axes to configure at which position with respect to these other options the tick mark should be shown. For instance, suppose we want tick marks in `our system` for the x-axis at the bottom and at the top. This corresponds to once setting the `left axis` to its minimal value and once to its maximal value:



```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    x axis={visualize ticks={direction axis=left axis, left axis={goto=min}},
            visualize ticks={direction axis=left axis, left axis={goto=max}},
    }
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=many},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

- In the above example, we may wish to shorten the ticks a bit at the bottom and at the top. For this, we use the `low` and `high` key:

  `/tikz/data visualization/low=`⟨*dimension*⟩ (no default)

  When used with the `visualize ticks` option, the `low` key contains a dimension that specifies the extend of the tick going "toward the minimum" of the direction axis. More precisely, when a tick mark is visualized, a unit tangent vector at the current data point in the direction of the `direction axis` is computed and this vector is multiplied by ⟨*dimension*⟩ to compute the start position of the tick line. The end position is given by this vector times the `high` value.

  Note that the ⟨*dimension*⟩ should usually be negative for the `low` key and positive for the `high` key.

  For tick marks where a tick label node is shown, the ⟨*dimension*⟩ is increased by the current values of keys like `tick text even low padding`, see Section 82.4.12 for details.

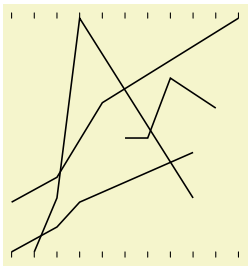
  `/tikz/data visualization/high=`⟨*dimension*⟩ (no default)

  Like `low`.

  `/tikz/data visualization/tick length=`⟨*dimension*⟩ (no default)

  Shorthand for `low=-`⟨*dimension*⟩`, high=`⟨*dimension*⟩.

  What we want to happen is that in the upper visualization of the ticks the `low` value is `0pt`, while in the lower one the `high` value is `0pt`:



```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    x axis={
      visualize ticks={direction axis=left axis,high=0pt,left axis={goto=min}},
      visualize ticks={direction axis=left axis,low=0pt,left axis={goto=max}},
    }
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=many},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

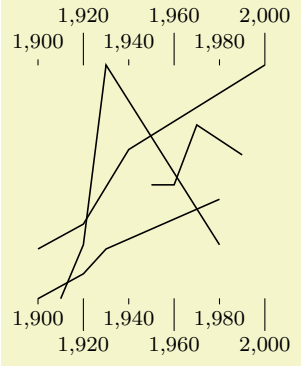
In order to style the tick mark, use the styling mechanism that is detailed in Section 82.4.9.

The tick label node. At certain tick positions, we may wish to add a node indicating the value of the attribute at the given position. The `visualize ticks` command has no influence over which text should be shown at a node – the text is specified and typeset as explained in Section 82.4.11.

Each time `visualize ticks`, for each tick position up to two tick label nodes will be created: One at the `low` position and one at the `high` position. The following keys are used to configure which of these cases happen:

`/tikz/data visualization/tick text at low=`⟨*true or false*⟩ (default `true`)

Pass this option to `visualize ticks` when you want tick label nodes to be placed at the `low` position of each tick mark.



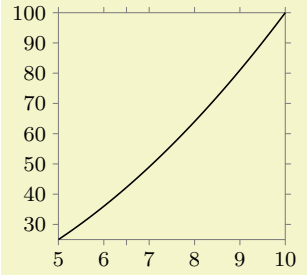

```
\usetikzlibrary {datavisualization}
\tikzset{
  data visualization/our system/.append style={
    x axis={
      visualize ticks={direction axis=left axis, left axis={goto=min},
                       high=0pt, tick text at low, stack},
      visualize ticks={direction axis=left axis, left axis={goto=max},
                       low=0pt, tick text at high, stack}
    }
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, length=3cm, ticks=some},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={people 1, people 2, money 1, money 2}]
  data group {people and money};
```

`/tikz/data visualization/tick text at high=`⟨*true or false*⟩ (default `true`)

Like `tick text at low`.

`/tikz/data visualization/no tick text` (no value)

Shorthand for `tick text at low=false, tick text at high=false`.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [scientific axes, all axes={length=3cm},
                          x axis={ticks={
                              major also at={6.5 as [no tick text]}}},
                          visualize as smooth line]
  data [format=function] {
    var x : interval [5:10];
    func y = \value x * \value x;
  };
```

When a tick label node is to be placed at the low or the high position, the next step is to determine the exact position and the correct anchor of the node. This is done as follows:

- In order to compute an appropriate `anchor`, the tick mark is considered: This is a short line pointing in a certain direction. For a tick label node at the `low` position, the `anchor` attribute is setup in such a way that the node label will be below the `low` position when the tick mark direction points up, it will be to the right when the direction points left, above when it points down, and so on also for diagonal directions. Similarly, for the `high` position, when the direction points up, the node will be placed above the tick mark and so on.

  This computation is done automatically.
- The tick label node is styled. The styles that are applied are described in Section 82.4.9.
- A tick label node for the `low` position is usually anchored at this `low` position, but an additional padding will be added as described in Section 82.4.12.

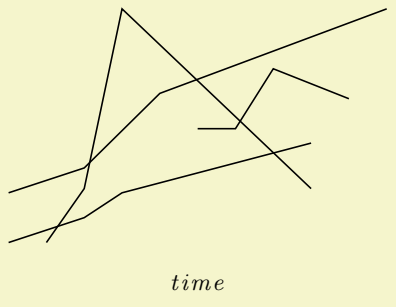
82.5.5 Visualizing the Axis Labels

The `label` option can be used with an axis to specify a text should be shown next to the axis to indicates which attribute this axis refers to. Like `ticks` or `grid`, the `label` option does not actually draw the label, this is the job of the `visualize label` key, which is configured by the axis system.

`/tikz/data visualization/axis options/visualize label=`⟨*options*⟩ (no default)

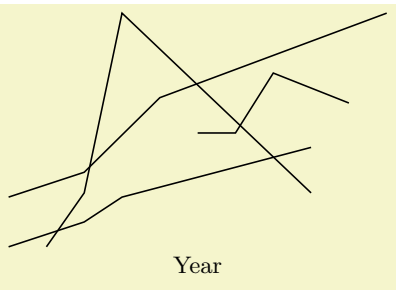
The ⟨*options*⟩ should be used to configure a "good place" for the axis label. Usually, you will use the `goto` or the `goto pos` key.

For the example of `our system`, we would like the label of the `x axis` to be placed below at the middle of the axis, so we use `goto pos=.5` to determine this position. Concerning the other axes, we want it to be placed at the minimum position of the `left axis` with a lot of padding.




```
\usetikzlibrary {datavisualization}
\tikzdatavisualizationset{
  our system/.append style={
    x axis={visualize label={
        x axis={goto pos=.5},
        left axis={padding=1.5em, goto=padded min}}}
  }
}
\tikz \datavisualization [
    our system,
    x axis={attribute=time, ticks=some, label},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={
      people 1, people 2, money 1, money 2}]
  data group {people and money};
```

In the above example, the `padding` of `1.5em` was rather arbitrary and "suboptimal". It would be outright wrong if the labels on the `x axis` were larger or if they were missing. It would be better if the vertical position of the `x axis` label were always "below" all other options. For such cases a slightly strange approach is useful: You position the node using `node style={at=...}` where `at` is now the normal Ti*k*Z option that is used to specify the position of a node. Inside the `...`, you specify that the horizontal position should be the bottom of up-to-now-constructed data visualization and the vertical position should be at the "origin", which is, however, the position computed by the `goto` keys for the axes:




```
\usetikzlibrary {datavisualization}
\tikzdatavisualizationset{
  our system/.append style={
    x axis={visualize label={
      x axis={goto pos=.5},
      node style={
        at={(0,0 |- data visualization bounding box.south)},
        below
} } } } }
\tikz \datavisualization [
    our system,
    x axis={attribute=time, ticks=some, label=Year},
    left axis ={attribute=money},
    right axis={attribute=people},
    visualize as line/.list={
      people 1, people 2, money 1, money 2}]
  data group {people and money};
```

Two additional keys are useful for positioning axis labels:

`/tikz/data visualization/axis option/anchor at min` (no value)

When passed to an axis, this key sets the `anchor` so that a node positioned at either the `min` or the `padded min` value of the axis will be placed "nicely" with respect to the axis. For instance, if the axis points upwards from the `min` value to the `max` value, the `anchor` would be set to `north` since this gives a label below the axis's start. Similarly, if the axis points right, the anchor would be set to `east`, and so on.

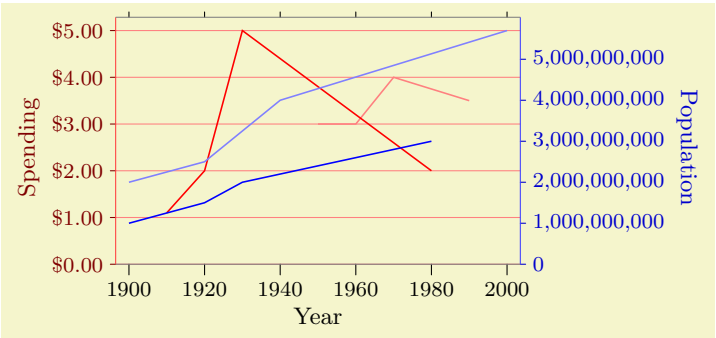
`/tikz/data visualization/axis option/anchor at max` (no value)

Like `anchor at min`.

82.5.6 The Complete Axis System

Here is the code for the complete axis system developed above and an example of how it is used:

```
\tikzdatavisualizationset{ our system/.style={
  % The axes
  new Cartesian axis=x axis,     new Cartesian axis=left axis,         new Cartesian axis=right axis,
  % The directions of the axes
  all axes={padding=.5em},       left axis={unit vector={(0cm,1pt)}},  right axis={unit vector={(0cm,1pt)}},
  % The default attributes, other attributes must be configured
  x axis={attribute=x},
  % The lengths of the axes
  x axis    ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/width}},
  left axis ={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
  right axis={length=\pgfkeysvalueof{/tikz/data visualization/scientific axes/height}},
  % The styling of the axes
  every axis/.style={style=black!50}, % make this the default
  % Visualizing the axes themselves
  left axis= {visualize axis={x axis=   {goto=padded min}, style=red!75, padded}},
  right axis={visualize axis={x axis=   {goto=padded max}, style=blue!75,padded}},
  x axis=    {visualize axis={left axis={goto=padded min}, padded},
              visualize axis={left axis={goto=padded max}, padded}},
  % Visualizing the grid, when requested
  x axis=    {visualize grid={direction axis=left axis}},
  left axis= {visualize grid={direction axis=x axis, common={style=red!50}}},
  right axis={visualize grid={direction axis=x axis, common={style=blue!50}}},
  % Visualizing the ticks, when requested
  left axis={visualize ticks={style={red!50!black}, direction axis=x axis,
                              x axis={goto=padded min}, high=0pt, tick text at low}},
  right axis={visualize ticks={style={blue!80!black}, direction axis=x axis,
                              x axis={goto=padded max}, low=0pt, tick text at high}},
  x axis={visualize ticks={direction axis=left axis, left axis={goto=padded min}, high=0pt,
                           tick text at low},
          visualize ticks={direction axis=left axis, left axis={goto=padded max}, low=0pt}},
  % By default, there are ticks on all axes
  all axes={ticks},
  % Visualizing the axis labels, when requested
  x axis={visualize label={x axis={goto pos=.5}, node style={
        at={(0,0 |- data visualization bounding box.south)}, below}}},
  left axis={visualize label={left axis={goto pos=.5}, node style={
        at={(0,0 -| data visualization bounding box.west)}, rotate=90, anchor=south, red!50!black}}},
  right axis={visualize label={right axis={goto pos=.5}, node style={
        at={(0,0 -| data visualization bounding box.east)}, rotate=-90, anchor=south,  blue!80!black}}},
}}
```



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  our system,
  x axis={attribute=time, label=Year,
    ticks={tick text padding=2pt,  style={/pgf/number format/set thousands separator={}}},
  left axis={attribute=money, label=Spending,
    padding min=0, include value=0, grid,
    ticks={tick prefix=\$, style={/pgf/number format/fixed,
        /pgf/number format/fixed zerofill, /pgf/number format/precision=2}},
  right axis={attribute=people,
    label=Population,
    padding min=0, include value=0,
    ticks={style=/pgf/number format/fixed}},
  visualize as line/.list={
    people 1, people 2, money 1, money 2},
  people 1={style={visualizer color=blue}},
  people 2={style={visualizer color=blue!50}},
  money 1={style={visualizer color=red}},
  money 2={style={visualizer color=red!50}} ]
data group {people and money};
```

82.5.7 Using the New Axis System Key

The axis system `our system` that we developed in the course of the previous section is not yet "configurable". The only configuration that was possible was to "misuse" the `width` and `height` keys of the `scientific axes`.

In order to make `our system` configurable so that we can say `our system=`⟨*options*⟩, where ⟨*options*⟩ are executed with the path prefix

```
/tikz/data visualization/our system
```

we can use the following key:

`/tikz/data visualization/new axis system=`{⟨*axis system name*⟩}{⟨*axis setup*⟩}{⟨*default options*⟩}{⟨*application options*⟩} (no default)

The `new axis system` key takes four parameters. The first one, ⟨*system name*⟩, is the name of the to-be-created axis system, `our system` in our case. The `new axis system` will create the following new key:

`/tikz/data visualization/`⟨*axis system name*⟩`=`⟨*options*⟩ (no default)

When the key ⟨*axis system name*⟩ is used, the following keys will be executed in the following order:

1. The ⟨*axis setup*⟩ with the path prefix `/tikz/data visualization/`.
2. The ⟨*default options*⟩ with the same path prefix.
3. The following style:

   `/tikz/data visualization/every` ⟨*axis system name*⟩ (style, no value)

   Even though this style has the path prefix `/tikz/data visualization` itself, the keys stored in this style will be executed with the path prefix `/tikz/data visualization/`⟨*axis system name*⟩.
4. The ⟨*options*⟩ with the path prefix `/tikz/data visualization/`⟨*axis system name*⟩.
5. The ⟨*application options*⟩ with the path prefix `/tikz/data visualization/`

Let us now have a look at what all of this means. First, the ⟨*axis setup*⟩ will contain all options that setup the axis system in all ways that need not be configured. For instance, the ⟨*axis setup*⟩ for the `scientific axes` will create an `x axis` and also a `y axis` (because these are always present), but will not setup the label visualization (because this can be configured in different ways). For `our system`, which cannot be configured at all, we would place all of our configuration in the ⟨*axis setup*⟩.

The ⟨*default options*⟩ can be used to pick default values that would usually be passed to the ⟨*options*⟩ of the newly created axis system. For instance, for `scientific axis`, the ⟨*default options*⟩ are set to `outer ticks,standard labels`, because these are the defaults.

Finally, the ⟨*application options*⟩ can be used to actually apply the configuration that has been chosen by the ⟨*options*⟩. The idea is that ⟨*default options*⟩, ⟨*options*⟩, and also `every` ⟨*axis system name*⟩ all have a chance of changing, re-changing and re-setting all sorts of styles and keys. Then, with the

last change “winning”, the resulting setting of a style can be executed, which may then cause a label visualization to be installed.

83 Visualizers

83.1 Overview

In a data visualization a long stream of data points is *visualized* using *visualizers*. Recall that it is the job of the axis systems as described in Section 82 to determine *where* data points are visualized. It is the job of the visualizers to determine *how* they are visualized.

The most basic and common visualizer is the *line visualizer*. It simply connects subsequent data points by straight lines to indicate either that the points on these lines interpolate between the real data points or the straight lines are used to indicate the order in which the data points appear. A different, more "conservative" visualizer is the *scatter visualizer* or *mark visualizer*, which just places a small mark at each data point. Such a visualizer does not imply any interpolation or ordering between the data points.

Visualizers may, however, also be more complicated. For instance, a visualizer used for a box plot could visualize a data point as a box with a median value, standard deviation, outliers, and other information; a rectangle visualizer might visualize data points as larger areas; a projection visualizer might visualize the projection of data points onto different axes; and so.

Creating a new visualizer is not quite trivial since a new PGF class needs to be implemented. Fortunately, using visualizers is much simpler: For each kind of visualizer there is a key that allows you to create such a visualizer. You can then use further keys to configure the visualizer and to connect it to the data.

In a data visualization multiple visualizers may exist at the same time. This happens in different situations:

- A data visualization may contain several independent data sets that are to be visualized. There might be a line plot, for which a line visualizer is used, and also a scatter plot, for which a scatter visualizer would be used.

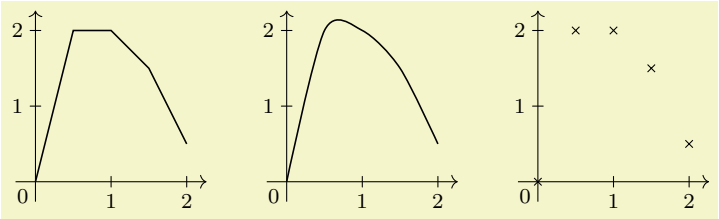
  In this case, for each data point only one visualizer will do anything. To achieve this, each data point has an attribute called `visualizer` which tells the visualizer objects whether they should "react" to the data point or not.

- A single data point might be visualized several times. For instance, a scatter visualizer might draw a mark at the data point's position on the page and a projection visualizer might draw, additionally, a mark at the projected position.

83.2 Usage

83.2.1 Using a Single Visualizer

The simplest scenario for using visualizers are data visualizations in which there is only a single data set that is visualized in one style. In this case, all that needs to be done in order to choose a visualizer is use one of the options starting with `visualize as ...` together with the `\datavisualization` command:



```
\usetikzlibrary {datavisualization}
% Define a data set:
\tikz \datavisualization data group {example} = {
data {
  x, y
  0, 0
  0.5, 2
  1, 2
  1.5, 1.5
  2, 0.5
}};
\tikz \datavisualization [school book axes, visualize as line]        data group {example};
\qquad
\tikz \datavisualization [school book axes, visualize as smooth line] data group {example};
\qquad
\tikz \datavisualization [school book axes, visualize as scatter]     data group {example};
```

Methods for styling visualizers are discussed in Section 83.2.3.

83.2.2 Using Multiple Visualizers

A data visualization may contain multiple data groups and for each data set we might wish to use a different visualizer. In this case, we need some way of telling the data visualization engine to which visualizer should be used with the different data points.

To solve this problem, you can *name* a visualizer. The visualizer's name can then both be used to configure the visualizer and also to indicate that data points "belong" to the visualizer.

Naming a visualizer is quite simple: The `visualize as ...` keys actually take a single parameter, which is the name of the visualizer. For instance, the following code creates three visualizers, named `sin`, `cos`, and `tan`:

```
visualize as line=sin,
visualize as line=cos,
visualize as scatter=tan
```

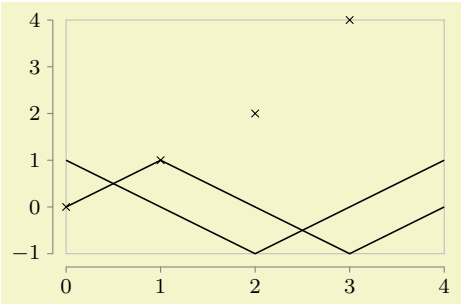
(When you just say `visualize as line` without providing a name, the name `line` is chosen as a default, for `visualize as scatter` the name `scatter` is the default and so.)

In order to indicate which data points should be visualized by which of these visualizers, the following key is important:

`/data point/set` (no value)

A visualizer will only act on a data point when its name matches the value of this key. Initially, this key is set to the last visualizer created, so if there is only one, there is no need to set or worry about this key.

Since the `set` key has the path prefix `/data point`, it can be set like any other attribute of a data key:



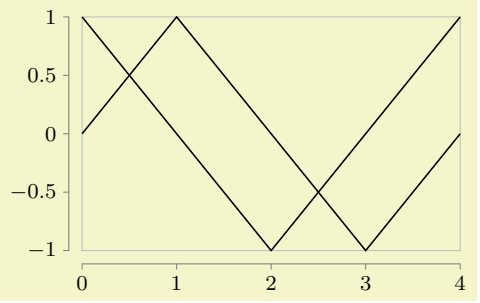

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as line=sin,
  visualize as line=cos,
  visualize as scatter=tan]
data {
  x, y, set
  0, 0, sin
  1, 1, sin
  2, 0, sin
  3, -1, sin
  4, 0, sin
  0, 1, cos
  1, 0, cos
  0, 0, tan
  1, 1, tan
  2, 2, tan
  3, 4, tan
  2, -1, cos
  3, 0, cos
  4, 1, cos
};
```

As can be seen, the data points with the same `set` attribute do not need to be consecutive.

The above method of specifying the visualizer works nicely, but in most cases it would be more natural to keep the `set` attribute out of the table. This is easy to achieve by using multiple `data` and using the following key:

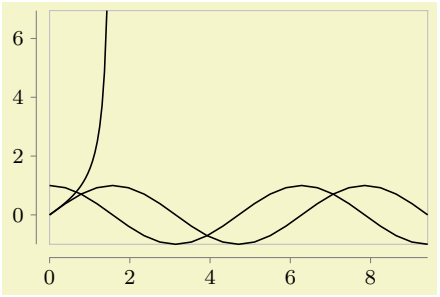
`/pgf/data/set`=⟨*name*⟩ (no default)

Shorthand for `/data point/set`=⟨*name*⟩.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as line=sin,
  visualize as line=cos]
data [set=sin] {
  x, y
  0, 0
  1, 1
  2, 0
  3, -1
  4, 0
}
data [set=cos] {
  x, y
  0, 1
  1, 0
  2, -1
  3, 0
  4, 1
};
```

When you need to visualize several similar things in a single plot (like ten lines that all get visualized by `visualize as line`), it is somewhat cumbersome having to write this ten times. In this case you can shorten your code by making use of the `.list` key handler: When you add it to a key, the "value" passed to the key is parsed as a list of values. The key is then executed once for each of these values:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as line/.list={sin, cos, tan}]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
}
data [set=tan, format=function] {
  var x : interval[0:pi/2.2];
  func y = tan(\value x r);
};
```

83.2.3 Styling a Visualizer

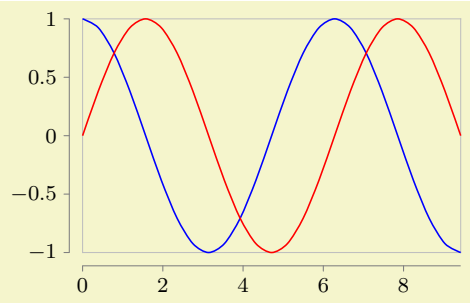
In order to style a visualizer that has been created using for instance `visualize as line`=⟨*visualizer name*⟩, you can use the following key:

`/tikz/data visualization/`⟨*visualizer name*⟩=⟨*options*⟩ (no default)

For each visualizer, a key of the same name is created with the path prefix `/tikz/data visualization`. This key takes the ⟨*options*⟩ and executes them with the path prefix

```
/tikz/data visualization/visualizer options/
```

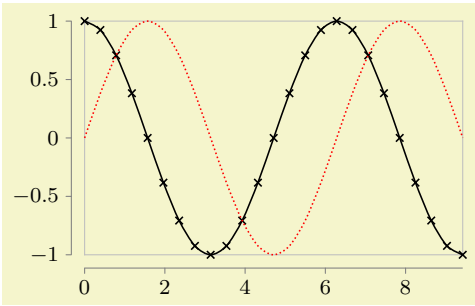
These options are then used to configure the appearance of the current visualizer. (This is quite similar to the way options are passed to an axis in order to configure the axis.) Possible options include `style`, but also `label in legend` and `label in data`. The latter two options are discussed in Section 84.3, the first option below.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as smooth line/.list={sin, cos},
  sin={style=red},
  cos={style=blue}]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
};
```

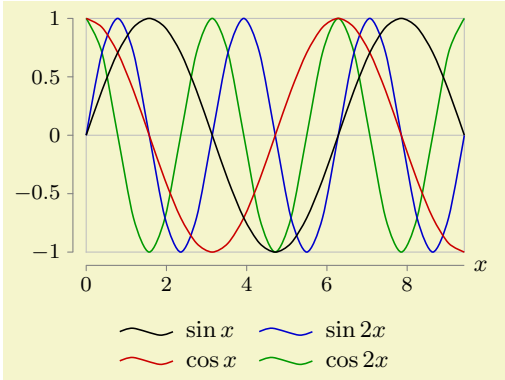
/tikz/data visualization/visualizer options/style=⟨*options*⟩ (no default)

The ⟨*options*⟩ given to this key should be normal Ti*k*Z options. They will be executed when the visualizer is used.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as smooth line=sin,
  sin={style={red, densely dotted}},
  visualize as smooth line=cos,
  cos={style={mark=x}},
]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
};
```

When you have multiple visualizers in a single data visualization, you can use the `style` option with each visualizer to configure their different appearances as in the above example. However, it is usually much better (and easier) to use a style sheet, see Section 84.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes={clean, end labels},
  x axis={label=$x$}, y axis={grid={major also at=0}},
  visualize as smooth line/.list={sin,cos,sin 2,cos 2},
  legend={below, rows=2},
  sin={label in legend={text=$\sin x$}},
  cos={label in legend={text=$\cos x$}},
  sin 2={label in legend={text=$\sin 2x$}},
  cos 2={label in legend={text=$\cos 2x$}},
  style sheet=strong colors]
data [set=sin, format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
}
data [set=cos, format=function] {
  var x : interval[0:3*pi];
  func y = cos(\value x r);
}
data [set=sin 2, format=function] {
  var x : interval[0:3*pi];
  func y = sin(2*\value x r);
}
data [set=cos 2, format=function] {
  var x : interval[0:3*pi];
  func y = cos(2*\value x r);
};
```

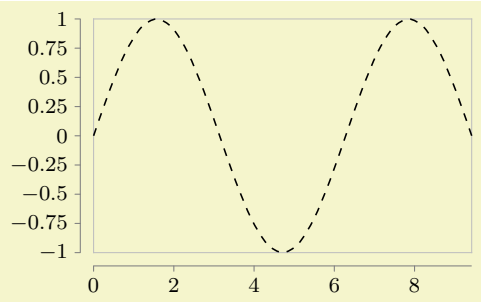
/tikz/data visualization/visualizer options/ignore style sheets (no value)

This option, which should be passed to a visualizer after its creation before another visualizer is created, causes style sheets *not* to apply to the visualizer (but the `style` option will still have an effect). This allows you to create visualizers that are used for special purposes and that do not "take part" in the usual styling. For instance, a visualizer might be used internally to depict a regression line, even though the regression line itself should not participate in the usual styling by, say, dashing or different coloring.

In addition to the options passed to a visualizer via `style`, the following also gets executed when a visualizer is used:

`/tikz/data visualization/every visualizer` (style, no value)

This style is used with every visualizer. Note that it should contain normal Ti*k*Z keys.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  every visualizer/.style={dashed},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:3*pi];
  func y = sin(\value x r);
};
```

83.3 Reference: Basic Visualizers

83.3.1 Visualizing Data Points Using Lines

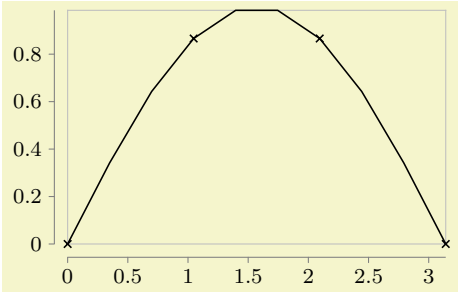
`/tikz/data visualizers/visualize as line=`⟨*visualizer name*⟩ (default `line`)

Creates a new visualizer named ⟨*visualizer name*⟩. Basically, this visualizer connects all data points for which the `/data point/set` attribute equals ⟨*visualizer name*⟩ by a line that is styled by the visualizer's style.

In more detail, the following happens:

1. A new object is created (of class `plot handler visualizer`) that is configured to collect the canvas positions of all data points whose `set` attribute equals ⟨*visualizer name*⟩.
2. During the end of the data visualization, PGF's plotting mechanism (see Section 112) is used to plot the stream of recorded data points.

   This means that, in principle, all of the plot handlers available in Ti*k*Z could be used for the visualization (such as the `smooth` handler). However, some plot handlers such as, say, the `xcomb` are unsuitable as plot handlers since they do not support the advanced axis handling done by the data visualization engine. Because of this (and also for other reasons), you cannot set the plot handler directly, but must use one of the options like `straight line`, `smooth line` and others, documented in a moment.
3. Additionally, plot marks can be drawn at the collected data points. Here, all of the options available to Ti*k*Z for drawing plot marks are available. To configure them, all options offered by Ti*k*Z for configuring marks are available such as `mark repeat`:

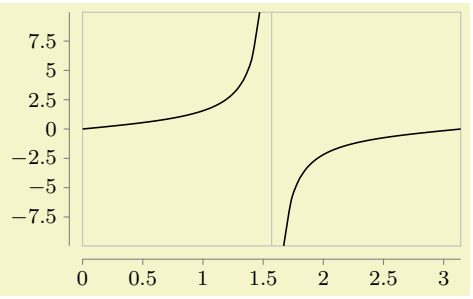



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as line=my data,
  my data={style={mark=x, mark repeat=3}}]
data [format=function] {
  var x : interval [0:pi] samples 10;
  func y = sin(\value x r);
};
```

The line visualizer also provides a method of dealing with gaps in a line. Take for instance the function $f(x) = \tan x$. When this function is plotted over the interval $[0, \pi]$, then the function will go to $\pm\infty$ at $\pi/2$. When we plot this, we might plot the function in the interval $[0, \frac{\pi}{2} - \epsilon]$ and then continue in the interval $[\frac{\pi}{2} + \epsilon, \pi]$. However, we do not want the point at coordinate $\big(\frac{\pi}{2} - \epsilon, \tan(\frac{\pi}{2} - \epsilon)\big)$ to be connected to the coordinate $\big(\frac{\pi}{2} + \epsilon, \tan(\frac{\pi}{2} + \epsilon)\big)$ by a line. Rather, there should be a "gap" or a "jump" between these coordinates. To achieve this, the following key can be used:

`/data point/outlier=`⟨*value*⟩ (default `true`, initially empty)

When this key is set to anything non-empty value, a visualizer will consider this data point to be an "outlier". For a line visualizer this means that the point is not shown and that the current line ends at the previous data point and a new line starts at the next data point.



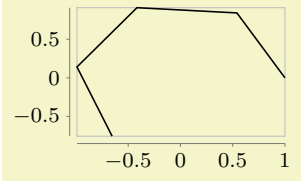

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean, x axis={grid={major at={(pi/2)}}},
  visualize as smooth line]
data [format=function] {
  var x : interval[0:pi/2-0.1];
  func y = tan(\value x r);
}
data point [outlier]
data [format=function] {
  var x : interval[pi/2+0.1:pi];
  func y = tan(\value x r);
};
```

`/tikz/data visualizers/visualize as smooth line=`⟨*visualizer name*⟩ (default `line`)

A shorthand `visualize as line=`⟨*visualizer name*⟩ followed ⟨*visualizer name*⟩`=smooth line`.

`/tikz/data visualization/visualizer options/straight line` (no value)

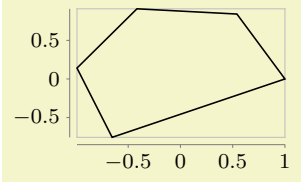
Causes the data points to be connected by straight lines.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={straight line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

`/tikz/data visualization/visualizer options/straight cycle` (no value)

Causes the data points to be connected by a polygon.



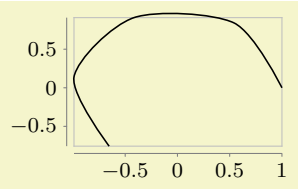

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={straight cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

`/tikz/data visualization/visualizer options/polygon` (no value)

This is an alias for `straight cycle`.

`/tikz/data visualization/visualizer options/smooth line` (no value)

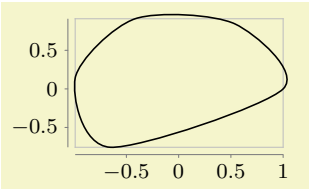
Causes the data points to be connected by a line that is smoothed at the joins:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={smooth line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/smooth cycle (no value)

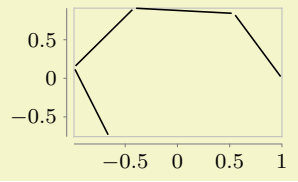
Causes the data points to be connected by a circular line that is smoothed at the joins:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={smooth cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap line (no value)

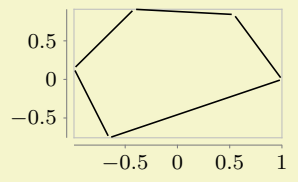
This key causes the data points to be connected by lines that "do not quite touch" the data points. This is implemented by using the `\pgfplothandlergaplineto`, see Section 65.5.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={gap line}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/gap cycle (no value)

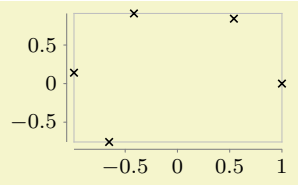
Like `gapped line`, only with a cycle:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={gap cycle}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

/tikz/data visualization/visualizer options/no lines (no value)

Suppresses the line. This option only makes sense when the `mark` option is used.

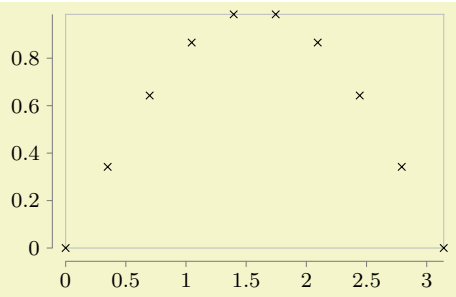



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz [scale=.55] \datavisualization
 [scientific axes=clean, all axes={ticks=few},
  visualize as smooth line=my data,  my data={no lines, style={mark=x}}]
data [format=function] {
  var t : interval [0:4] samples 5;
  func x = cos(\value t r);
  func y = sin(\value t r);
};
```

83.3.2 Visualizing Data Points Using Marks

`/tikz/data visualizers/visualize as scatter=`⟨*visualizer name*⟩ (default `scatter`)

A shorthand `visualize as line=`⟨*visualizer name*⟩ followed ⟨*visualizer name*⟩`=no lines` and setting the `style` of the visualizer so that is will use `mark=x` (plus some size adjustments) to draw marks at the data points.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization
 [scientific axes=clean,
  visualize as scatter]
data [format=function] {
  var x : interval [0:pi] samples 10;
  func y = sin(\value x r);
};
```

83.4 Advanced: Creating New Visualizers

Creating a new visualizer is a two-stage process that does, unfortunately, require in-depth knowledge of the data visualization backend:

1. First, you need to create a new class using `\pgfooclass` whose instances react to the signal `visualize datapoint signal`. This requires detailed knowledge of the data visualization engine, see Section 86.
2. Second, you should provide keys on the Ti*k*Z level for creating the necessary objects. These keys invoke the key `new visualizer` internally.

`/tikz/data visualization/new visualizer={`⟨*name*⟩`}{`⟨*options*⟩`}{`⟨*legend entry options*⟩`}` (no default)

This key configures a new visualizer named ⟨*name*⟩. This entails the following actions:

- The key `/tikz/data visualization/`⟨*name*⟩ is created. As described earlier, this key can be used to pass for instance `style` options to the visualizer.
- The style key `/tikz/data visualization/visualizers/`⟨*name*⟩`/styling` is created and made empty. This is the key in which the `style` key will store the options passed to the visualizer.
- The style key `/tikz/data visualization/visualizers/`⟨*name*⟩`/label in legend options` is set to ⟨*legend entry options*⟩. These options are used to configure how the visualizer should be rendered in a legend, see Section 84.9.9 for details.
- The key `/data point/set/`⟨*name*⟩ is set to a number that is increased for each visualizer in the current data visualization. This number is important for style sheets, see Section 84.
- The key `/data point/`⟨*name*⟩`/execute at begin` is set to code that creates a `{scope}` that executes the following styles as options:
  1. The ⟨*options*⟩ passed to the `new visualizer` key.
  2. The `every visualizer` style.
  3. The styling from the currently active style sheets, see Section 84.
  4. The styling stored in the `styling` key mentioned above.
- The key `/data point/`⟨*name*⟩`/execute at end` is set to code that will finish all paths that may have been created by the visualizer and closes the scope.

All of the above mean the following in practice:

- Inside a new `visualize as ...` key, you pass the name of the to-be-created to `new visualizer` as the first parameter and any special default styling setup of the visualizer as the second parameter.
- The new `visualize as ...` key should also create a visualizer object using `new object`.
- When this object finally is about to create the actual visualization, it should surround the code by invoking the code stored in the `execute at begin` and the `execute at end` keys of the visualizer.

Everything else is usually taken care of by the `new visualizer` key automatically.

As an example, let us create a simple visualizer that creates a circle whose radius is dictated by the `radius` attribute. To keep things simple in this example, this attribute cannot be configured.

First, we need the visualizer class. For this example I have boiled it down to a minimum:

```
\pgfooclass{circle visualizer}
{
  % Stores the name of the visualizer. This is needed for filtering and configuration
  \attribute name;

  % The constructor. Just setup the attribute.
  \method circle visualizer(#1) { \pgfooset{name}{#1} }

  % Connect to visualize signal.
  \method default connects() {
    \pgfoothis.get handle(\me)
    \pgfkeysvalueof{/pgf/data visualization/obj}.connect(\me,visualize,visualize datapoint signal)
  }

  % This method is invoked for each data point. It checks whether the data point belongs to the correct
  % visualizer and, if so, calls the macro \dovisualization to do the actual visualization.
  \method visualize() {
    \pgfdvfilterpassedtrue
    \pgfdvnamedvisualizerfilter
    \ifpgfdvfilterpassed
      \dovisualization
    \fi
  }
}
```

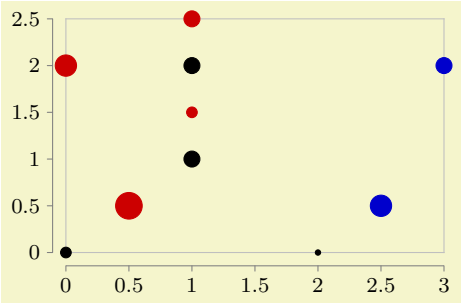
The `\dovisualization` method must now do the correct visualization.

```
\def\dovisualization{
  \pgfkeysvalueof{/data point/\pgfoovalueof{name}/execute at begin}
    \pgfpathcircle{\pgfpointdvdatapoint}{\pgfkeysvalueof{/data point/radius}}
    % \pgfusepath is done by |execute at end|
  \pgfkeysvalueof{/data point/\pgfoovalueof{name}/execute at end}
}
```

Finally, we create a `visualize as` key:

```
\tikzdatavisualizationset{
  visualize as circle/.style={
    new object={
      when=after survey,
      store=/tikz/data visualization/visualizers/#1,
      class=circle visualizer,
      arg1=#1
    },
    new visualizer={#1}{%
      color=visualizer color,        % a color setup by the style sheet
      every path/.style={fill,draw}, % fill and draw the circle by default,
    }{}, % let's ignore legends in this example
    /data point/set=#1
  },
  visualize as circle/.default=circle
}
```

Now, let's see how this works:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  scientific axes=clean,
  visualize as circle/.list={a, b, c},
  style sheet=strong colors]
data [set=a] {
  x, y, radius
  0, 0, 2pt
  1, 1, 3pt
  1, 2, 3pt
  2, 0, 1pt
}
data [set=b] {
  x, y, radius
  0.5, 0.5, 5pt
  1, 1.5, 2pt
  1, 2.5, 3pt
  0, 2, 4pt
}
data [set=c] {
  x, y, radius
  3, 2, 3pt
  2.5, 0.5, 4pt
};
```

84 Style Sheets and Legends

84.1 Overview

In many data visualizations, different sets of data need to be visualized in a single visualization. For instance, in a plot there might be a line for the sine of x and another line for the cosine of x; in another visualization there might be a set of points representing data from a first experiment and another set of points representing data from a second experiment; and so on. In order to indicate to which data set a data point belongs, one might plot the curve of the sine in, say, black, and the curve of the cosine in red; we might plot the data from the first experiment using stars and the data from the second experiment using circles; and so on. Finally, at some place in the visualization – either inside the data or in a legend next to it – the meaning of the colors or symbols need to be explained.

Just as you would like Ti*k*Z to map the data points automatically onto the axes, you will also typically wish Ti*k*Z to choose for instance the coloring of the lines automatically for you. This is done using *style sheets*. There are at least two good reasons why you should prefer style sheets over configuring the styling of each visualizer "by hand" using the `style` key:

1. It is far more convenient to just say `style sheet=strong colors` than having to individually picking the different colors.

2. The style sheets were chosen and constructed rather carefully.

   For instance, the `strong colors` style sheet does not pick colors like pure green or pure yellow, which have very low contrast with respect to a white background and which often lead to unintelligible graphics. Instead, opposing primary colors with maximum contrast on a white background were picked that are visually quite pleasing.

   Similarly, the different dashing style sheets are constructed in such a way that there are only few and small gaps in the dashing so that no data points get lost because the dashes are spaced too far apart. Also dashing patterns were chosen that have a maximum optical difference.

   As a final example, style sheets for plot marks are constructed in such a way that even when two plot marks lie directly on top of each other, they are still easily distinguishable.

The bottom line is that whenever possible, you should use one of the predefined style sheets rather than picking colors or dashings at random.

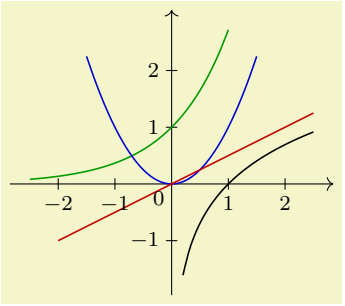
84.2 Concepts: Style Sheets

A *style sheet* is a predefined list of styles such as a list of colors, a list of dashing pattern, a list of plot marks, or a combinations thereof. A style sheet can be *attached* to a data point attribute. Then, the value of this attribute is used with data points to choose which style in the list should be chosen to visualize the data point.

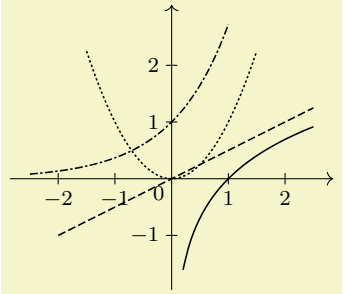
In most cases, there is just one attribute to which style sheets get attached: the `/data point/visualizer` attribute. The effect of attaching a style sheet to this attribute is that each visualizer is styled differently.

For the following examples, let us first define a simple data set:

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization data group {function classes} = {
  data [set=log, format=function] {
    var x : interval [0.2:2.5];
    func y = ln(\value x);
  }
  data [set=lin, format=function] {
    var x : interval [-2:2.5];
    func y = 0.5*\value x;
  }
  data [set=squared, format=function] {
    var x : interval [-1.5:1.5];
    func y = \value x*\value x;
  }
  data [set=exp, format=function] {
    var x : interval [-2.5:1];
    func y = exp(\value x);
  }
};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=strong colors]
data group {function classes};
```



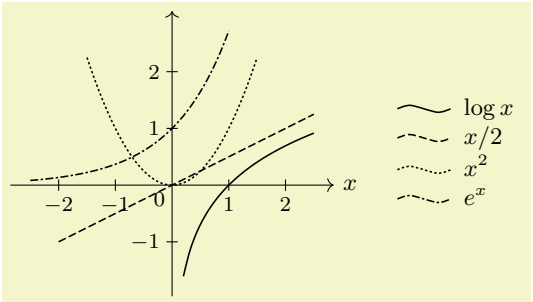

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=vary dashing]
data group {function classes};
```

84.3 Concepts: Legends

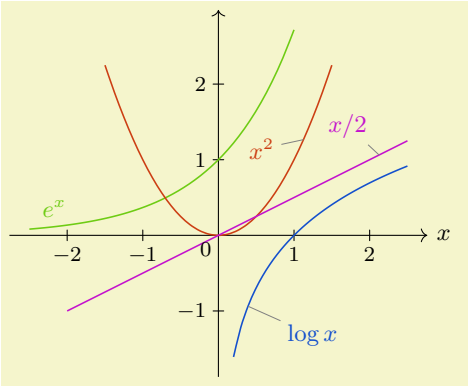
A *legend* is a box that is next to a data visualization (or inside it at some otherwise empty position) that contains a textual explanation of the different colors or styles used in a data visualization.

Just as it is difficult to get colors and dashing patterns right "by hand", it is also difficult to get a legend right. For instance, when a small line is shown in the legend that represents the actual line in the data visualization, if the line is too short and the dashing is too large, it may be impossible to discern which dashing is actually meant. Similarly, when plot marks are shown on such a short line, using a simple straight line may make it hard to read the plot marks correctly.

The data visualization engine makes some effort to make it easy to create high-quality legends. Additionally, it also offers ways of easily adding labels for visualizers directly inside the data visualization, which is even better than adding a legend, in general.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log=     {pin in data={text'=$\log x$, when=y is -1}},
  lin=     {pin in data={text=$x/2$, when=x is 2,
                         pin length=1ex}},
  squared={pin in data={text=$x^2$, when=x is 1.1,
                         pin angle=230}},
  exp=     {label in data={text=$e^x$, when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

84.4 Usage: Style Sheets

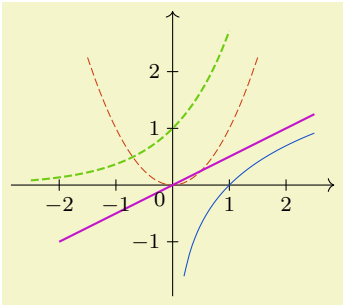
84.4.1 Picking a Style Sheet

To use a style sheet, you need to *attach* it to an attribute. You can attach multiple style sheets to an attribute and in this case all of these style sheets can influence the appearance of the data points.

Most of the time, you will attach a style sheet to the `set` attribute. This has the effect that each different data set inside the same visualization is rendered in a different way. Since this use of style sheets is the most common, there is a special, easy-to-remember option for this:

`/tikz/data visualization/style sheet`=⟨*style sheet*⟩ (no default)

Adds the ⟨*style sheet*⟩ to the list of style sheets attached to the `set` attribute.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes, all axes={unit length=7.5mm},
  visualize as smooth line/.list={log, lin, squared, exp},
  style sheet=vary thickness and dashing,
  style sheet=vary hue]
data group {function classes};
```

While the `style sheet` key will attach a style sheet only to the `set` attribute, the following key handler can be used to attach a style sheet to an arbitrary attribute:

Key handler ⟨*key*⟩`/.style sheet`=⟨*style sheet*⟩

Inside a data visualization you can use this key handler together with an attribute, that is, with a key having the path prefix `/data point`. For instance, in order to attach the ⟨*style sheet*⟩ `strong colors` to the attribute `set`, you could write

```
/data point/set/.style sheet=strong colors
```

Indeed, the `style sheet` key is just a shorthand for the above.

The effect of attaching a style sheet is the following:

- A new object is created that will monitor the attribute.
- Each time a special *styling key* is emitted by the data visualization engine, this object will inspect the current value of the attribute to which it is attached.
- Depending on this value, one of the styles stored in the style sheet is chosen (how this works, exactly, will be explained in a moment).
- The chosen style is then locally applied.

In reality, things are a bit more complicated: If the attribute of the data point happens to have a subkey named in the same way as the value, then the value of is this subkey is used instead of the value itself. This allows you to "rename" a value.

In a sense, a style sheet behaves much like a visualizer (see Section 83): In accordance with the value of a certain attribute, the appearance of data points change. However, there are a few differences: First, the styling of a data point needs to be triggered explicitly and this triggering is not necessarily done for each data point individually, but only for a whole visualizer. Second, styles can be computed even when no data point is present. This is useful for instance in a legend since, here, a visual representation of a visualizer needs to be created independently of the actual data points.

84.4.2 Creating a New Style Sheet

Creating a style sheet works as follows: For each possible value that an attribute can attain we must specify a style. This is done by creating a style key for each such possible value with a special path prefix and setting this style key to the desired value. The special path prefix is `/pgf/data visualization/style sheets` followed by the name of the style sheet.

As an example, suppose we wish to create a style sheet `test` that makes styled data points `red` when the attribute has value `foo` and `green` when the attribute has value `bar` and `dashed, blue` when the attribute is `foobar`. We could then write

```
/pgf/data visualization/style sheets/test/foo/.style={red},
/pgf/data visualization/style sheets/test/bar/.style={green},
/pgf/data visualization/style sheets/test/foobar/.style={dashed, blue},
```

We could then attach this style sheet to the attribute `code` as follows:

```
/data point/code/.style sheet=test
```

Then, when `/data point/code=foobar` holds when the styling signal is raised, the style `dashed, blue` will get executed.

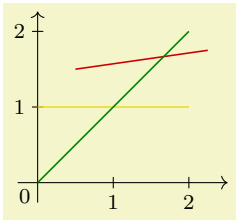
A natural question arises concerning the situation that the value of the attribute is not defined as a subkey of the style sheet. In this case, a special key gets executed:

`/pgf/data visualization/style sheets/`⟨*style sheet*⟩`/default style=`⟨*value*⟩ (style, no default)

This key gets during styling whenever `/pgf/data visualization/style sheet/`⟨*style sheet*⟩`/`⟨*value*⟩ is not defined.

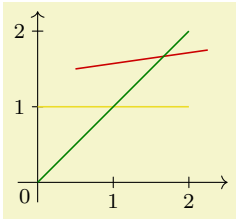
Let us put all of this together in a real-life example. Suppose we wish to create a style sheet that makes the first data set `green`, the second `yellow` and the third one `red`. Further data sets should be, say, `black`. The attribute that we intend to style is the `set` attribute. For the moment, we assume that the data sets will be named `1`, `2`, `3`, and so on (instead of, say, `experiment 1` or `sin` or something more readable – we will get rid of this restriction in a minute).

We would now write:



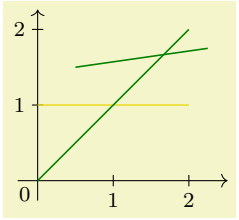

```
\usetikzlibrary {datavisualization}
\pgfkeys{
  /pgf/data visualization/style sheets/traffic light/.cd,
  % All these styles have the above prefix.
  1/.style={green!50!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
}
\tikz \datavisualization [
  school book axes,
  visualize as line=1,
  visualize as line=2,
  visualize as line=3,
  style sheet=traffic light]
data point [x=0, y=0, set=1]
data point [x=2, y=2, set=1]
data point [x=0, y=1, set=2]
data point [x=2, y=1, set=2]
data point [x=0.5, y=1.5, set=3]
data point [x=2.25, y=1.75, set=3];
```

In the above example, we have to name the visualizers `1`, `2`, `3` and so one since the value of the `set` attribute is used both assign data points to visualizers and also pick a style sheet. However, it would be much nicer if we could name any way we want. To achieve this, we use the special rule for style sheets that says that if there is a subkey of an attribute whose name is the same name as the value, then the value of this key is used instead. This slightly intimidating definition is much easier to understand when we have a look at an example:




```
\usetikzlibrary {datavisualization}
% Definition of traffic light keys as above
\begin{tikzpicture}
  \datavisualization data group {lines} = {
    data point [x=0, y=0,       set=normal]
    data point [x=2, y=2,       set=normal]
    data point [x=0, y=1,       set=heated]
    data point [x=2, y=1,       set=heated]
    data point [x=0.5, y=1.5,   set=critical]
    data point [x=2.25, y=1.75, set=critical]
  };
  \datavisualization [
    school book axes,
    visualize as line=normal,
    visualize as line=heated,
    visualize as line=critical,
    /data point/set/normal/.initial=1,
    /data point/set/heated/.initial=2,
    /data point/set/critical/.initial=3,
    style sheet=traffic light]
  data group {lines};
\end{tikzpicture}
```

Now, it is a bit bothersome that we have to set all these `/data point/set/...` keys by hand. It turns out that this is not necessary: Each time a visualizer is created, a subkey of `/data point/set` with the name of the visualizer is created automatically and a number is stored that is increased for each new visualizer in a data visualization. This means that the three lines starting with `/data point` are inserted automatically for you, so they can be left out. However, you would need them for instance when you would like several different data sets to use the same styling:




```
\usetikzlibrary {datavisualization}
% Definition of traffic light keys as above
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  /data point/set/critical/.initial=1, % same styling as first set
  style sheet=traffic light]
data group {lines};
```

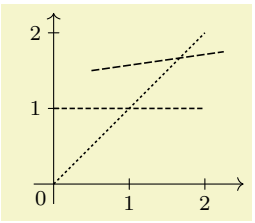
We can a command that slightly simplifies the definition of style sheets:

`\pgfdvdeclarestylesheet{`⟨*name*⟩`}{`⟨*keys*⟩`}`

This command executes the ⟨*keys*⟩ with the path prefix `/pgf/data visualization/style sheets/`⟨*name*⟩. The above definition of the traffic light style sheet could be rewritten as follows:

```
\pgfdvdeclarestylesheet{traffic light}{
  1/.style={green!50!black},
  2/.style={yellow!90!black},
  3/.style={red!80!black},
  default style/.style={black}
}
```

As a final example, let us create a style sheet that changes the dashing pattern according to the value of the attribute. We do not need to define an large number of styles in this case, but can use the `default style` key to "calculate" the correct dashing.



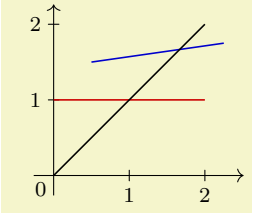

```
\usetikzlibrary {datavisualization}
\pgfdvdeclarestylesheet{my dashings}{
  default style/.style={dash pattern={on #1pt off 1pt}}
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my dashings]
data group {lines};
```

84.4.3 Creating a New Color Style Sheet

Creating a style sheet that varies colors according to an attribute works the same way as creating a normal style sheet: Subkeys lies `1`, `2`, and so on use the `style` attribute to setup a color. However, instead of using the `color` attribute to set the color, you should use the `visualizer color` key to set the color:

`/tikz/visualizer color`=⟨*color*⟩ (no default)

This key is used to set the color `visualizer color` to ⟨*color*⟩. This color is used by visualizers to color the data they visualize, rather than the current "standard color". The reason for not using the normal current color is simply that it makes many internals of the data visualization engine a bit simpler.

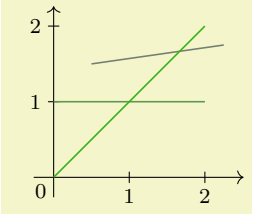



```
\usetikzlibrary {datavisualization}
\pgfdvdeclarestylesheet{my colors}
{
  default style/.style={visualizer color=black},
  1/.style={visualizer color=black},
  2/.style={visualizer color=red!80!black},
  3/.style={visualizer color=blue!80!black},
}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=my colors]
data group {lines};
```

There is an additional command that makes it easy to define a style sheet based on a *color series*. Color series are a concept from the `xcolor` package: The idea is that we start with a certain color for the first data set and then add a certain "color offset" for each next data point. Please consult the documentation of the `xcolor` package for details.

`\tikzdvdeclarestylesheetcolorseries`{⟨*name*⟩}{⟨*color model*⟩}{⟨*initial color*⟩}{⟨*step*⟩}

This command creates a new style sheet using `\pgfdvdeclarestylesheet`. This style sheet will only have a default style setup that maps numbers to the color in the color series starting with ⟨*initial color*⟩ and having a stepping of ⟨*step*⟩. Note that when the value of the attribute is `1`, which it is the first data set, the *second* color in the color series is used (since counting starts at `0` for color series). Thus, in general, you need to start the ⟨*initial color*⟩ "one early".




```
\usetikzlibrary {datavisualization}
\tikzdvdeclarestylesheetcolorseries{greens}{hsb}{0.3,1.3,0.8}{0,-.4,-.1}
\tikz \datavisualization [
  school book axes,
  visualize as line=normal,
  visualize as line=heated,
  visualize as line=critical,
  style sheet=greens]
data group {lines};
```

84.5 Reference: Style Sheets for Lines

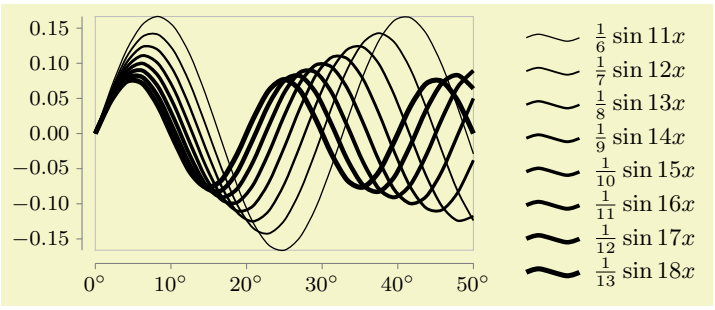
The following style sheets can be applied to visualizations that use the `visualize as line` and related keys. For the examples, the following style and data set are used:

```
\tikzdatavisualizationset {
  example visualization/.style={
    scientific axes=clean,
    y axis={ticks={style={
          /pgf/number format/fixed,
          /pgf/number format/fixed zerofill,
          /pgf/number format/precision=2}}},
    x axis={ticks={tick suffix=${}^\circ$}},
    1={label in legend={text=$\frac{1}{6}\sin 11x$}},
    2={label in legend={text=$\frac{1}{7}\sin 12x$}},
    3={label in legend={text=$\frac{1}{8}\sin 13x$}},
    4={label in legend={text=$\frac{1}{9}\sin 14x$}},
    5={label in legend={text=$\frac{1}{10}\sin 15x$}},
    6={label in legend={text=$\frac{1}{11}\sin 16x$}},
    7={label in legend={text=$\frac{1}{12}\sin 17x$}},
    8={label in legend={text=$\frac{1}{13}\sin 18x$}}
  }
}
```

```
\tikz \datavisualization data group {sin functions} = {
  data [format=function] {
    var set : {1,...,8};
    var x : interval [0:50];
    func y = sin(\value x * (\value{set}+10))/(\value{set}+5);
  }
};
```

Style sheet `vary thickness`

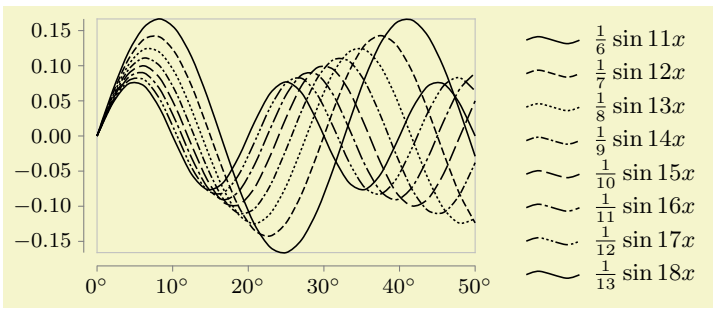
This style varies the thickness of lines. It should be used only when there are only two or three lines, and even then it is not particularly pleasing visually.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness]
data group {sin functions};
```

Style sheet `vary dashing`

This style varies the dashing of lines. Although it is not particularly pleasing visually and although visualizations using this style sheet tend to look "excited" (but not necessarily "exciting"), this style sheet is often the best choice when the visualization is to be printed in black and white.

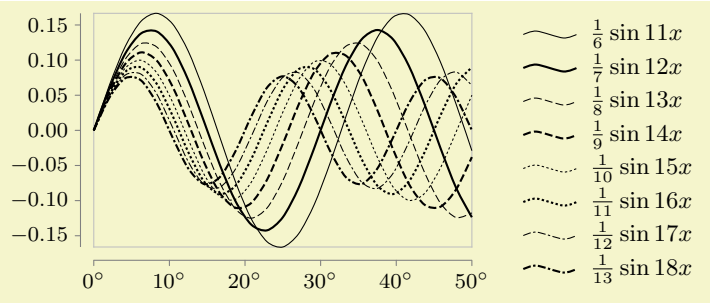



```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary dashing]
data group {sin functions};
```

As can be seen, there are only seven distinct dashing patterns. The eighth and further lines will use a solid line once more. You will then have to specify the dashing "by hand" using the `style` option together with the visualizer.

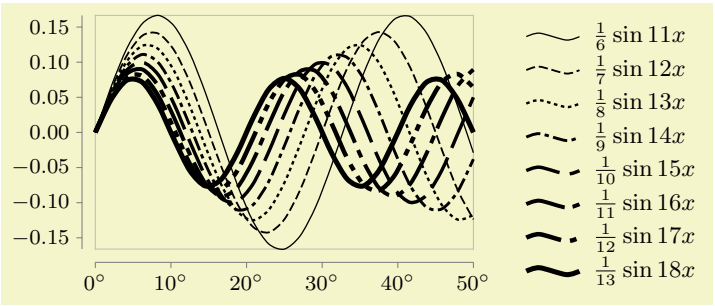
Style sheet `vary thickness and dashing`

This style alternates between varying the thickness and the dashing of lines. The difference to just using both the `vary thickness` and `vary dashing` is that too thick lines are avoided. Instead, this style creates clearly distinguishable line styles for many lines (up to 14) with a minimum of visual clutter. This style is the most useful for visualizations when many different lines (ten or more) should be printed in black and white.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness
              and dashing]
data group {sin functions};
```

For comparison, here is the must-less-than-satisfactory result of combining the two independent style sheets:




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary thickness,
  style sheet=vary dashing]
data group {sin functions};
```

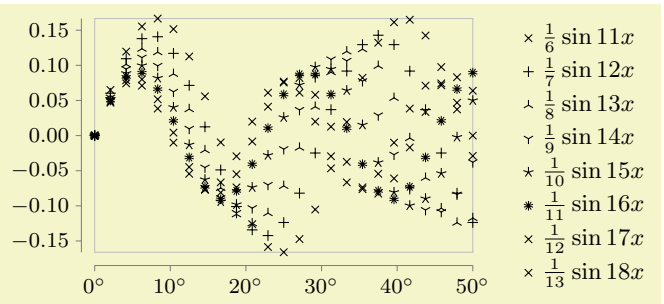
84.6 Reference: Style Sheets for Scatter Plots

The following style sheets can be used both for scatter plots and also with lines. In the latter case, the marks are added to the lines.

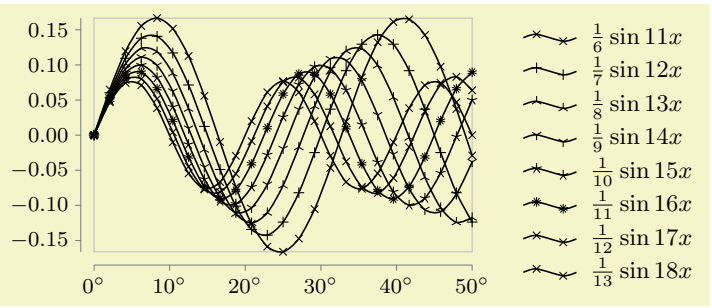
Style sheet `cross marks`

This style uses different crosses to distinguish between the data points of different data sets. The crosses were chosen in such a way that when two different cross marks lie at the same coordinate, their overall shape allows one to still uniquely determine which marks are on top of each other.

This style supports only up to six different data sets and requires the `plotmarks` library.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as scatter/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=cross marks]
data group {sin functions};
```




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=cross marks]
data group {sin functions};
```

84.7 Reference: Color Style Sheets

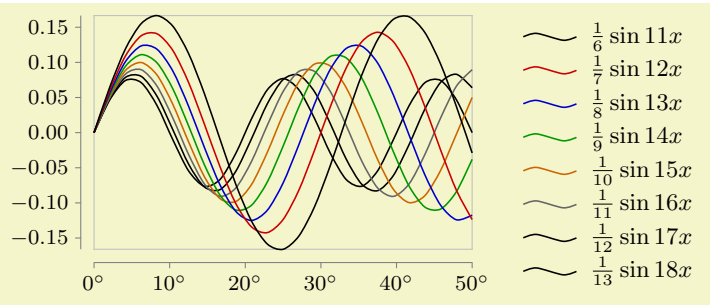
Color style sheets are very useful for creating visually pleasing data visualizations that contain multiple data sets. However, there are two things to keep in mind:

- At some point, every data visualization is printed or photo copied in black and white by someone. In this case, data sets can often no longer be distinguished.
- A few people are color blind. They will not be able to distinguish between red and green lines (and some people are not even able to distinguish colors at all).

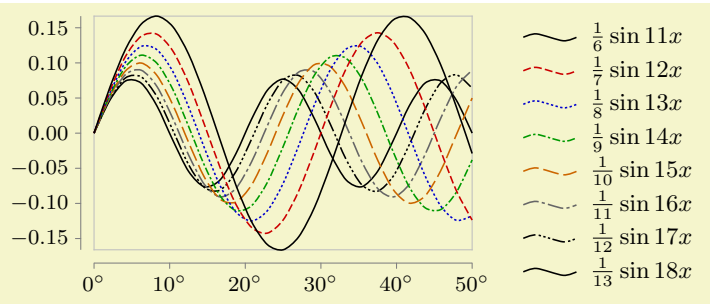
For these reasons, if there is any chance that the data visualization will be printed in black and white at some point, consider combining color style sheets with style sheets like `vary dashing` to make data sets distinguishable in all situations.

Style sheet `strong colors`

This style sheets uses pure primary colors that can very easily be distinguished. Although not as visually pleasing as the `vary hue` style sheet, the visualizations are easier to read when this style sheet is used. Up to six different data sets are supported.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=strong colors]
data group {sin functions};
```

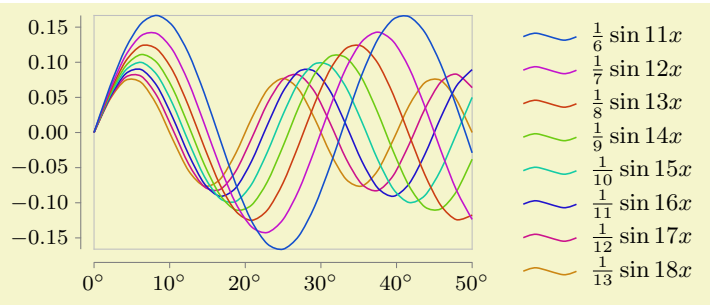



```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=strong colors,
  style sheet=vary dashing]
data group {sin functions};
```

Unlike `strong colors`, the following style sheets support, in principle, an unlimited number of data set. In practice, as always, more than four or five data sets lead to nearly indistinguishable data sets.

Style sheet `vary hue`

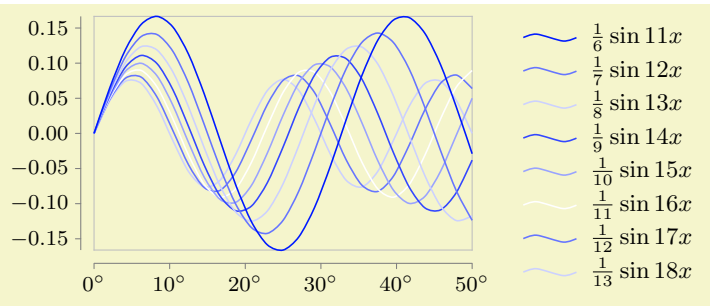
This style uses a different hue for each data set.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=vary hue]
data group {sin functions};
```

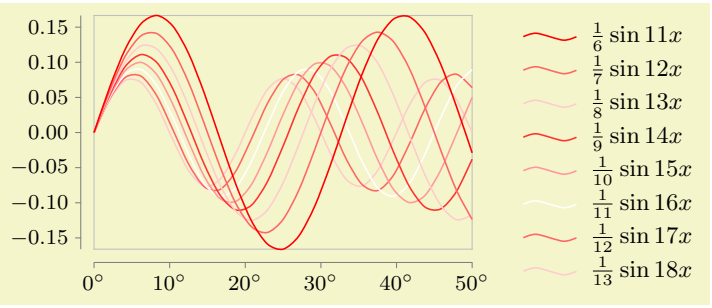
Style sheet `shades of blue`

As the name suggests, different shades of blue are used for different data sets.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=shades of blue]
data group {sin functions};
```

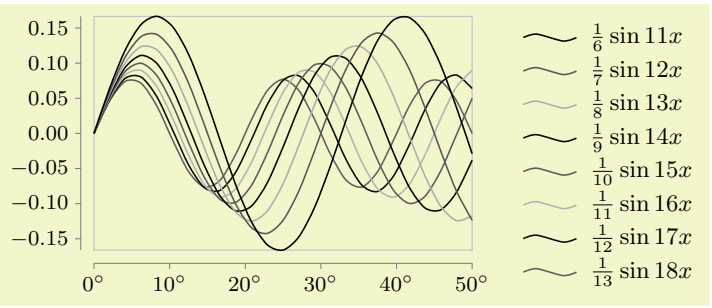
Style sheet `shades of red`




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=shades of red]
data group {sin functions};
```

Style sheet `gray scale`

For once, this style sheet can also be used when the visualization is printed in black and white.




```
\usetikzlibrary
{datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list=
    {1,2,3,4,5,6,7,8},
  example visualization,
  style sheet=gray scale]
data group {sin functions};
```

84.8 Usage: Labeling Data Sets Inside the Visualization

In a visualization that contains multiple data sets, it is often necessary to clearly point out which line or mark type corresponds to which data set. This can be done in the main text via a sentence like "the normal data (black) lies clearly below the critical values (red)", but it often a good idea to indicate data sets ideally directly inside the data visualization or directly next to it in a so-called legend.

The data visualization engine has direct support both for indicating data sets directly inside the visualization and also for indicating them in a legend.

The "best" way of indicating where a data set lies or which color is used for it is to put a label directly inside the data visualization. The reason this is the "best" way is that people do not have to match the legend entries against the data, let alone having to look up the meaning of line styles somewhere in the text. However, adding a label directly inside the visualization is also the most tricky way of indicating data sets since it is hard to compute good positions for the labels automatically and since there needs to be some empty space where the label can be put.

84.8.1 Placing a Label Next to a Data Set

The following key is used to create a label inside the data visualization for a data set:

`/tikz/data visualization/visualizer options/label in data`=⟨*options*⟩ (no default)

This key is passed to a visualizer that has previously been created using keys starting `visualize as ...`. It will create a label inside the data visualization "next" to the visualizer (the details are explained in a moment). You can use this key multiple times with a visualizer to create multiple labels at different points with different texts.

The ⟨*options*⟩ determine which text is shown and where it is shown. They are executed with the following path prefix:

```
/tikz/data visualization/visualizer label options
```

In order to configure which text is shown and where, use the following keys inside the ⟨*options*⟩:

`/tikz/data visualization/visualizer label options/text`=⟨*text*⟩ (no default)

This is the text that will be displayed next to the data. It will be to the "left" of the data, see the description below.

`/tikz/data visualization/visualizer label options/text'`=⟨*text*⟩ (no default)

Like `text`, only the text will be to the "right" of the data.

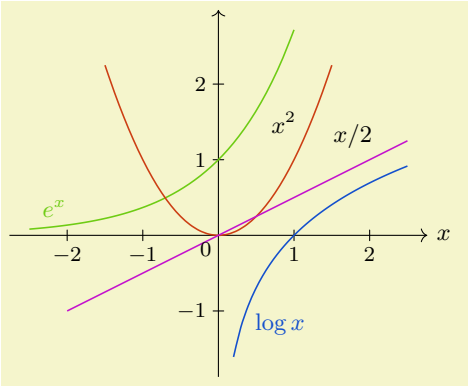
The following keys are used to configure where the label will be shown. They use different strategies to specify one data point where the label will be anchored. The coordinate of this data point will be stored in `(label visualizer coordinate)`. Independently of the strategy, once the data point has been chosen, the coordinate of the next data point is stored in `(label visualizer coordinate')`. Then, a (conceptual) line is created from the first coordinate to the second and a node is placed at the beginning of this line to its "left" or, for the `text'` option, on its "right". More precisely, an automatic anchor is computed for a node placed implicitly on this line using the `auto` option or, for the `text'` option, using `auto,swap`.

The node placed at the position computed in this way will have the ⟨*text*⟩ set by the `text` or `text'` option and its styling is determined by the current `node style`.

Let us now have a look at the different ways of determining the data point at which the label in anchored:

`/tikz/data visualization/visualizer label options/when`=⟨*attribute*⟩ `is`⟨*number*⟩ (no default)

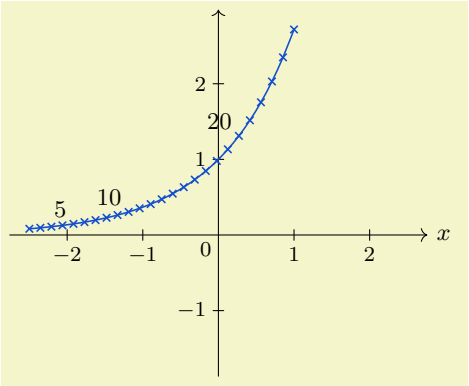
This key causes the value of the ⟨*attribute*⟩ to be monitored in the stream of data points. The chosen is data point is the first data point where the ⟨*attribute*⟩ is at least ⟨*number*⟩ (if this never happens, the last data point is used).




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  log=     {label in data={text'=$\log x$, when=y is -1,
                           text colored}},
  lin=     {label in data={text=$x/2$,     when=x is 2}},
  squared={label in data={text=$x^2$,     when=x is 1.1}},
  exp=     {label in data={text=$e^x$,     when=x is -2,
                           text colored}},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/index=⟨*number*⟩ (no default)

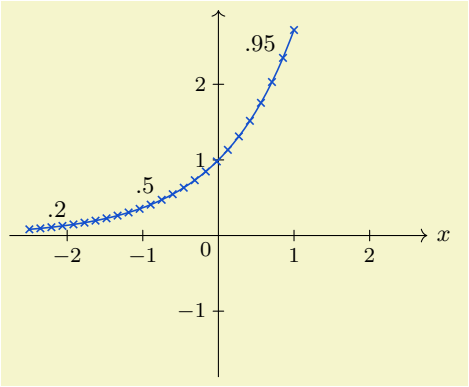
This key chooses the ⟨*number*⟩th data point belonging to the visualizer's data set.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={exp},
  exp=     {label in data={text=$5$, index=5},
            label in data={text=$10$, index=10},
            label in data={text=$20$, index=20},
            style={mark=x}},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/pos=⟨*fraction*⟩ (no default)

This key chooses the first data point belonging to the data set whose index is at least ⟨*fraction*⟩ times the number of all data points in the data set.

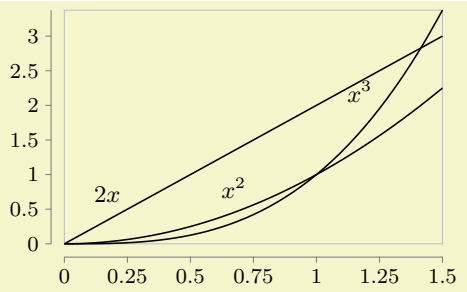



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line=exp,
  exp=     {label in data={text=$.2$, pos=0.2},
            label in data={text=$.5$, pos=0.5},
            label in data={text=$.95$, pos=0.95},
            style={mark=x}},
  style sheet=vary hue]
data group {function classes};
```

/tikz/data visualization/visualizer label options/auto (no value)

This key is executed automatically by default. It works like the `pos` option, where the ⟨*fraction*⟩ is set to (⟨*data set's index*⟩ − 1/2)/⟨*number of data sets*⟩. For instance, when there are 10 data sets, the fraction for the first one will be 5%, the fraction for the second will be 15%, for the third it will be 25%, ending with 95% for the last one.

The net effect of all this is that when there are several lines, labels will be placed at different positions along the lines with hopefully only little overlap.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line/.list={linear, squared, cubed},
  linear ={label in data={text=$2x$}},
  squared={label in data={text=$x^2$}},
  cubed  ={label in data={text=$x^3$}}]
data [set=linear, format=function] {
  var x : interval [0:1.5];
  func y = 2*\value x;
}
data [set=squared, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x;
}
data [set=cubed, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x * \value x;
};
```

As can be seen in the example, the result is not always satisfactory. In this case, the `pin in data` option might be preferable, see below.

The following keys allow you to style labels.

`/tikz/data visualization/visualizer label options/node style`=⟨*options*⟩ (no default)

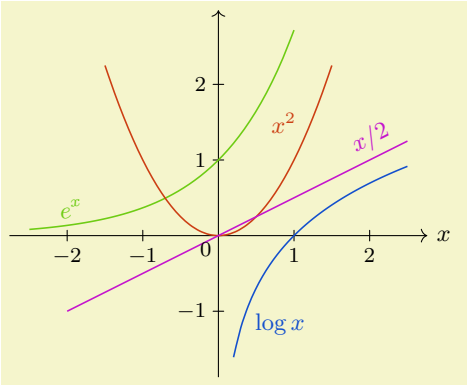
Just passes the options to `/tikz/data visualization/node style`.

`/tikz/data visualization/visualizer label options/text colored` (no value)

Causes the `node style` to set the text color to `visualizer color`. The effect of this is that the label's text will have the same color as the data set to which it is attached.

`/tikz/data visualization/every data set label` (style, no value)

This style is executed with every label that represents a data set. Inside this style, use `node style` to change the appearance of nodes. This style has a default definition, usually you should just append things to this style.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log=     {label in data={text'=$\log x$, when=y is -1}},
  lin=     {label in data={text=$x/2$,
                           node style=sloped,    when=x is 2}},
  squared={label in data={text=$x^2$,     when=x is 1.1}},
  exp=     {label in data={text=$e^x$,
                           node style=sloped,    when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

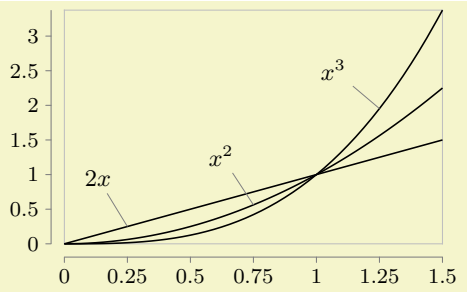
`/tikz/data visualization/every label in data` (style, no value)

Like `every data set label`, this key is also executed with labels. However, this key is executed after the style sheets have been executed, giving you a chance to overrule their styling.

84.8.2 Connecting a Label to a Data Set via a Pin

`/tikz/data visualization/visualizer options/pin in data`=⟨*options*⟩ (no default)

This key is a variant of the `label in data` key and takes the same options, plus two additional ones. The difference to `label in data` is that the label node is shown a bit removed from the data set, but connected to it via a small line (this is like the difference between the `label` and `pin` options).




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes=clean,
  visualize as smooth line/.list={linear, squared, cubed},
  linear ={pin in data={text=$2x$}},
  squared={pin in data={text=$x^2$}},
  cubed  ={pin in data={text=$x^3$}}]
data [set=linear, format=function] {
  var x : interval [0:1.5];
  func y = \value x;
}
data [set=squared, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x;
}
data [set=cubed, format=function] {
  var x : interval [0:1.5];
  func y = \value x * \value x * \value x;
};
```

The following keys can be used additionally:

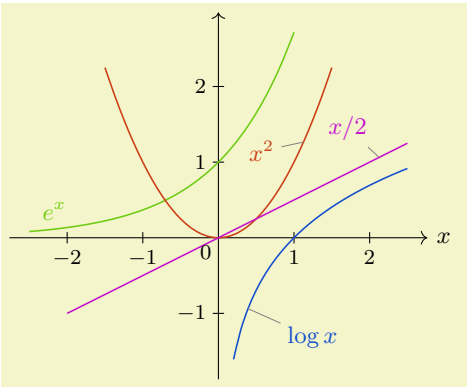
`/tikz/data visualization/visualizer label options/pin angle`=⟨*angle*⟩ (no default)

The position of the label of a `pin in data` is mainly computed in the same way as for a `label in data`. However, once the position has been computed, the label is shifted as follows:

- When an ⟨*angle*⟩ is specified using the present key, the shift is by the current value of `pin length` in the direction of ⟨*angle*⟩.
- When ⟨*angle*⟩ is empty (which is the default), then the shift is also by the current value of `pin length`, but now in the direction that is orthogonal and to the left of the line between the coordinate of the data point and the coordinate of the next data point. When `text'` is used, the direction is to the right instead of the left.

`/tikz/data visualization/visualizer label options/pin length`=⟨*dimension*⟩ (no default)

See the description of `pin angle`.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  school book axes,
  x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  every data set label/.append style={text colored},
  log=     {pin in data={text'=$\log x$, when=y is -1}},
  lin=     {pin in data={text=$x/2$, when=x is 2,
                         pin length=1ex}},
  squared={pin in data={text=$x^2$, when=x is 1.1,
                         pin angle=230}},
  exp=     {label in data={text=$e^x$, when=x is -2}},
  style sheet=vary hue]
data group {function classes};
```

84.9 Usage: Labeling Data Sets Inside a Legend

The "classical" way of indicating the style used for the different data sets inside a visualization is a *legend*. It is a description next to or even inside the visualization that contains one line for each data set and displays an iconographic version of the data set next to some text labeling the data set. Note, however, that even though legend are quite common, also consider using a `label in data` or a `pin in data` instead.

Creating a high-quality legend is by no means simple. A legend should not distract the reader, so aggressive borders should definitively be avoided. A legend should make it easy to match the actual styling of a data set (like, say, using a red, dashed line) to the "iconographic" representation of this styling. An example of what can go wrong here is using short lines to represent lines dashed in different way where the lines are so short that the differences in the dashing cannot be discerned. Another example is showing straight lines with plot marks on them where the plot marks are obscured by the horizontal line itself, while the plot marks are clearly visible in the actual visualization since no horizontal lines occur.

The data visualization engine comes with a large set of options for creating and placing high-quality legends next or inside data visualizations.

84.9.1 Creating Legends and Legend Entries

A data visualization can be accompanied by one or more legends. In order to create a legend, the following key can be used (although, in practice, you will usually use the `legend` key instead, see below):

`/tikz/data visualization/new legend=`⟨*legend name*⟩ (default `main legend`)

This key is used to create a new legend named ⟨*legend name*⟩. The legend is empty by default and further options are needed to add entries to it. When the key is called a second time for the same ⟨*legend name*⟩ nothing happens.

When a legend is created, a new key is created that can subsequently be used to configure the legend:

`/tikz/data visualization/`⟨*legend name*⟩`=`⟨*options*⟩ (no default)

When this key is used, the ⟨*options*⟩ are executed with the path prefix

```
/tikz/data visualization/legend options
```

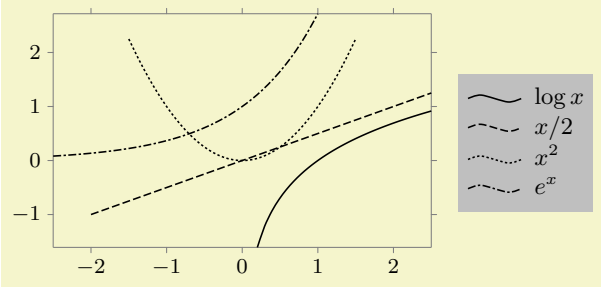
The different keys with this path prefix allow you to change the position where the legend is shown and how it is organised (for instance, whether legend entries are shown in a row or in a column or in a square).

The different possible keys will be explained in the course of this section.

In the end, the legend is just a Ti*k*Z node, a `matrix` node, to be precise. The following key is used to style this node:

`/tikz/data visualization/legend options/matrix node style=`⟨*options*⟩ (no default)

Adds the ⟨*options*⟩ to the list of options that will be executed when the legend's node is created.

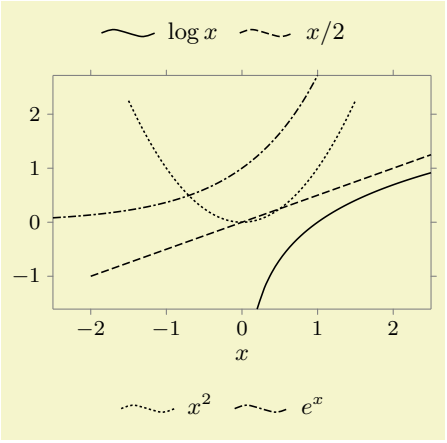



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={matrix node style={fill=black!25}},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

The following style allows you to configure the default appearance of every newly created legend:

`/tikz/data visualization/legend options/every new legend` (style, no value)

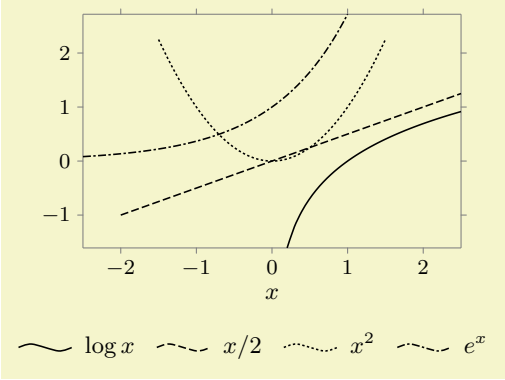
This key defaults to `east outside, label style=text right`. This means that by default a legend is placed to the right of the data visualization and that in the individual legend entries the text is to the right of the data set visualization.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  new legend={upper legend},
  new legend={lower legend},
  upper legend=above,
  lower legend=below,
  log=     {label in legend={text=$\log x$, legend=upper legend}},
  lin=     {label in legend={text=$x/2$, legend=upper legend}},
  squared={label in legend={text=$x^2$, legend=lower legend}},
  exp=     {label in legend={text=$e^x$, legend=lower legend}},
  style sheet=vary dashing]
data group {function classes};
```

`/tikz/data visualization/legend=`⟨*options*⟩ (no default)

This is a shorthand for `new legend=main legend, main legend=`⟨*options*⟩. In other words, this key creates a new `main legend` and immediately passes the configuration ⟨*options*⟩ to this legend.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list={log, lin, squared, exp},
  legend=below,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

As pointed out above, a legend is empty by default. In particular, the different data sets are not automatically inserted into the legend. Instead, the key `label in legend` must be used together with a data set:

`/tikz/data visualization/visualizer options/label in legend=`⟨*options*⟩ (no default)

This key is passed to a data set, similar to options like `pin in data` or `smooth line`. The ⟨*options*⟩ are used to configure the following:

- The legend in which the data set should be visualized.
- The text that is to be shown in the legend for the data set.
- The appearance of the legend entries.

In detail, the ⟨*options*⟩ are executed with the path prefix

```
/tikz/data visualization/legend entry options
```

To configure in which legend the label should appear, use the following key:

`/tikz/data visualization/legend entry options/legend=`⟨*name*⟩ (no default, initially `main legend`)

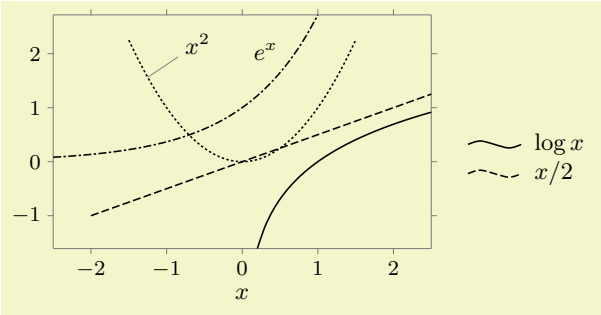
Set this key to the name of a legend that has previously been created using `new legend`. The label will then be shown in this legend.

In most cases, there is only one legend (namely `main legend`) and there is no need to set this key since it defaults to the main legend.

Also note that the legend ⟨*name*⟩ is automatically created if it nodes not yet exist.

`/tikz/data visualization/legend entry options/text`=⟨*text*⟩ (no default)

Use this key to setup the ⟨*text*⟩ that is shown as the label of the data set.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list=
  {log, lin, squared, exp},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={pin in data     ={text=$x^2$, pos=0.1}},
  exp=     {label in data  ={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

In addition to the two keys described above, there are further keys that are described in Section 84.9.6.

84.9.2 Rows and Columns of Legend Entries

In a legend, the different legend entries are arranged in a matrix, which typically has only one row or one column. For the impatient reader: Say `rows=1` to get everything in a row, say `columns=1` to get everything in a single column, and skip the rest of this section.

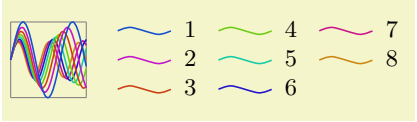
The more patient reader will appreciate that when there are very many different data sets in a single visualization, it may be necessary to use more than one row or column inside the legend. Ti*k*Z comes with a rather powerful mechanism for distributing the multiple legend entries over the matrix.

The first thing to decide is in which "direction" the entries should be inserted into the matrix. Suppose we have a 3×3 matrix and our entries are a, b, c, and so on. Then, one might place the a in the upper left corner of the matrix, b in the upper middle position, c in the upper right position, and d in the middle left position. This is a "first right, then down" strategy. A different strategy might be to place the a in the upper left corner, but b in the middle left position, c in the lower left position, and d then in the upper middle position. This is a "first down, then right" strategy. In certain situations it might even make sense to place a in the lower right corner and then go "first up, then left".

All of these strategies are supported by the `legend` command. You can configure which strategy is used using the following keys:

`/tikz/data visualization/legend options/down then right` (no value)

Causes the legend entries to fill the legend matrix first downward and, once a column is full, the next column is begun to the right of the previous one. This is the default.

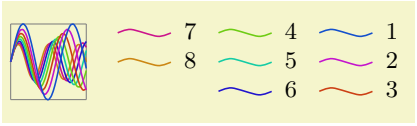



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={down then right, columns=3}]
data group {sin functions};
```

In the example, the `legend example` is the following style:

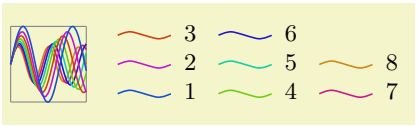
```
\tikzdatavisualizationset {
  legend example/.style={
    scientific axes, all axes={length=1cm, ticks=none},
    1={label in legend={text=1}},
    2={label in legend={text=2}},
    3={label in legend={text=3}},
    4={label in legend={text=4}},
    5={label in legend={text=5}},
    6={label in legend={text=6}},
    7={label in legend={text=7}},
    8={label in legend={text=8}}
  }
}
```

/tikz/data visualization/legend options/down then left (no value)



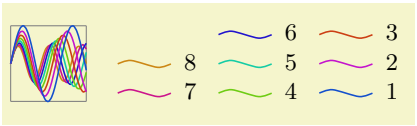

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={down then left, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/up then right (no value)



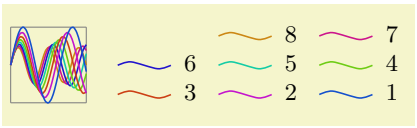

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={up then right, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/up then left (no value)



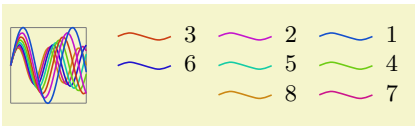

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={up then left, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/left then up (no value)



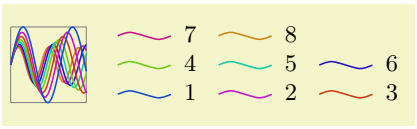

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={left then up, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/left then down (no value)



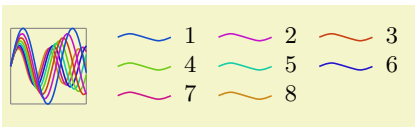

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={left then down, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/right then up (no value)




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then up, columns=3}]
data group {sin functions};
```

/tikz/data visualization/legend options/right then down (no value)

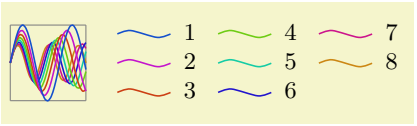



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down, columns=3}]
data group {sin functions};
```

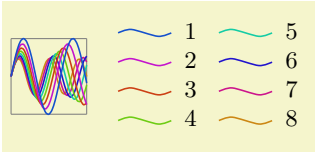
Having configured the directions in which the matrix is being filled, you must next setup the number of rows or columns that are to be shown. There are actually two different ways of doing so. The first way is to specify a maximum number of rows or columns. For instance, you might specify that there should be at most ten rows to a column and when there are more, a new column should be begun. This is achieved using the following keys:

`/tikz/data visualization/legend options/max rows`=⟨*number*⟩ (no default)

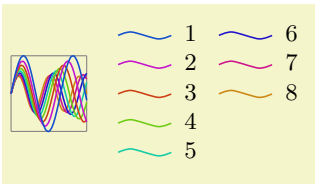
As the legend matrix is being filled, whenever the number of rows in the current column would exceed ⟨*number*⟩, a new column is started.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max rows=3}]
data group {sin functions};
```



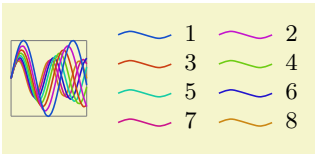

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max rows=4}]
data group {sin functions};
```



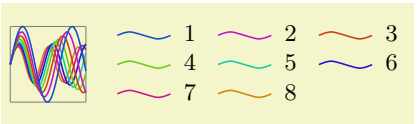

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max rows=5}]
data group {sin functions};
```

`/tikz/data visualization/legend options/max columns`=⟨*number*⟩ (no default)

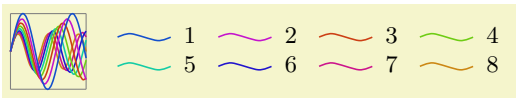
This key works like `max rows`, only now the number of columns is monitored. Note that this strategy only really makes sense when the when you use this key with a strategy that first goes left or right and then up or down.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down, max columns=2}]
data group {sin functions};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down,max columns=3}]
data group {sin functions};
```



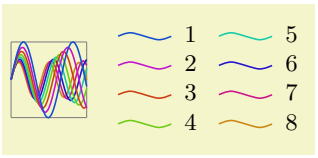

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={right then down,max columns=4}]
data group {sin functions};
```

The second way of specifying the number of entries in a row or column is to specify an "ideal number of rows or columns". The idea is as follows: Suppose that we use the standard strategy and would like to have everything in two columns. Then if there are eight entries, the first four should go to the first column,

while the next four should go to the second column. If we have 20 entries, the first ten should go the first column and the next ten to the second, and so on. So, in general, the objective is to distribute the entries evenly so the this "ideal number of columns" is reached. Only when there are too few entries to achieve this or when the number of entries per column would exceed the `max rows` value, will the number of columns deviate from this ideal value.

`/tikz/data visualization/legend options/ideal number of columns`=⟨*number*⟩ (no default)

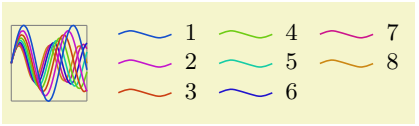
Specifies, that the entries should be split into ⟨*number*⟩ different columns, whenever possible. However, when there would be more than the `max rows` value of rows per column, more columns than the ideal number are created.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of columns=2}]
data group {sin functions};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of columns=4}]
data group {sin functions};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max rows=3,ideal number of columns=2}]
data group {sin functions};
```

`/tikz/data visualization/legend options/rows`=⟨*number*⟩ (no default)

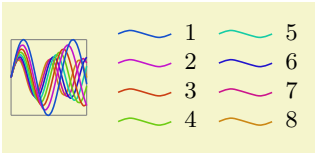
Shorthand for `ideal number of rows`=⟨*number*⟩.

`/tikz/data visualization/legend options/ideal number of rows`=⟨*number*⟩ (no default)

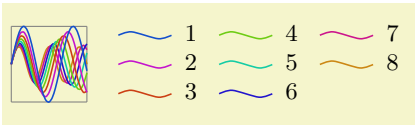
Works like `ideal number of columns`.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of rows=2}]
data group {sin functions};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={ideal number of rows=4}]
data group {sin functions};
```




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  visualize as smooth line/.list={1,2,3,4,5,6,7,8},
  legend example, style sheet=vary hue,
  main legend={max columns=3,ideal number of rows=2}]
data group {sin functions};
```

`/tikz/data visualization/legend options/columns`=⟨*number*⟩ (no default)

Shorthand for `ideal number of columns`=⟨*number*⟩.

84.9.3 Legend Placement: The General Mechanism

A legend can either be placed next to the data visualization or inside the data visualization at some place where there are no data entries. Both approached have advantages: Placing the legend next to the visualization minimises the "cluttering" by keeping all the extra information apart from the actual data, while placing the legend inside the visualization minimises the distance between the data sets and their explanations, making it easier for the eye to connect them.

For both approaches there are options that make the placement easier, see Sections 84.9.4 and 84.9.5, but these options internally just map to the following two options:

`/tikz/data visualization/legend options/anchor=`⟨*anchor*⟩ (no default)

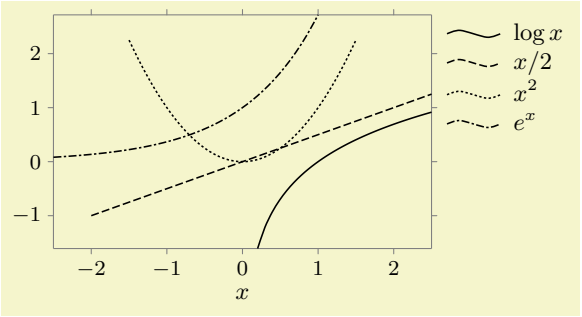
The whole legend is a Ti*k*Z-matrix internally. Thus, in particular, it is stored in a node, which has anchors. Like for any other node, when the node is shown, the node is shifted in such a way that the ⟨*anchor*⟩ of the node lies at the current `at` position.

`/tikz/data visualization/legend options/at=`⟨*coordinate*⟩ (no default)

Configures the ⟨*coordinate*⟩ at which the ⟨*anchor*⟩ of the legend's node should lie.

It may seem hard to predict a good ⟨*coordinate*⟩ for a legend since, depending of the size of the axis, different positions need to the chosen for the legend. However, it turns out that one can often use the coordinates of the special nodes `data bounding box` and `data visualization bounding box`, documented in Section 80.6.

As an example, let us put a legend to the right of the visualization, but so that the first entry starts at the top of the visualization:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes, x axis={label=$x$},
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={anchor=north west, at=
    (data visualization bounding box.north east)},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=vary dashing]
data group {function classes};
```

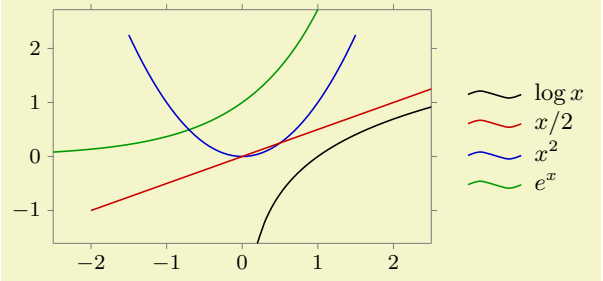
As can be seen, a bit of an additional shift might have been in order, but the result is otherwise quite satisfactory.

84.9.4 Legend Placement: Outside to the Data Visualization

The following keys make it easy to place a legend outside the data visualization.

`/tikz/data visualization/legend options/east outside` (no value)

Placing the legend to the right of the data visualization is the default:



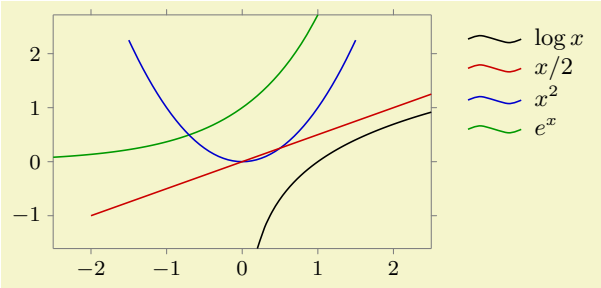

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=east outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

`/tikz/data visualization/legend options/right` (no value)

This is an easier-to-remember alias.

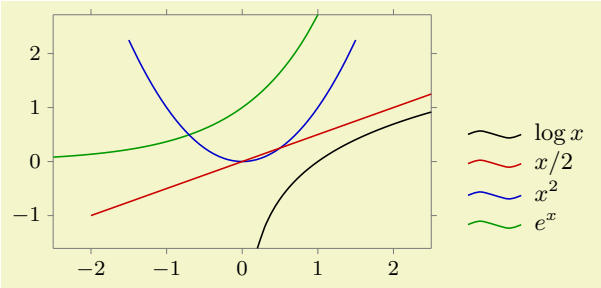
/tikz/data visualization/legend options/north east outside (no value)

A variant, where the legend is to the right, but aligned with the northern end of the data visualization:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=north east outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

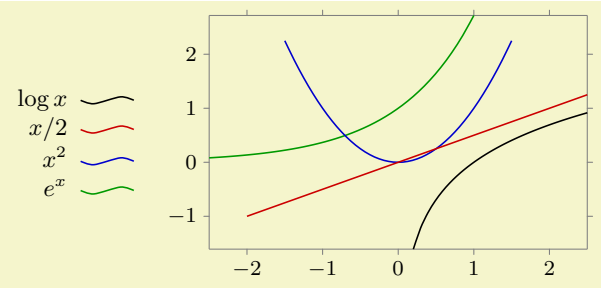
/tikz/data visualization/legend options/south east outside (no value)




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=south east outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/west outside (no value)

The legend is placed left. Note that the text also swaps its position.

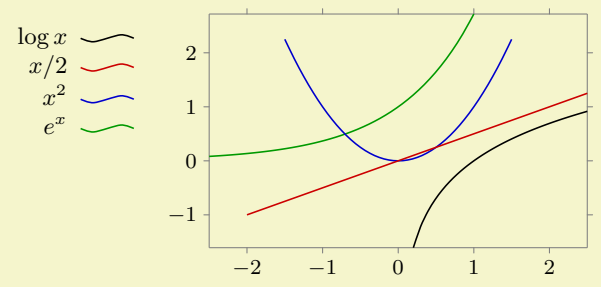



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=west outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/left (no value)

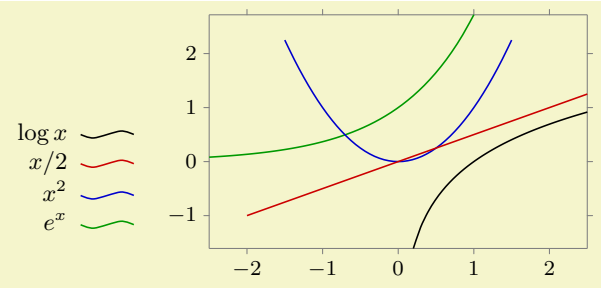
This is an easier-to-remember alias.

/tikz/data visualization/legend options/north west outside (no value)




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=north west outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

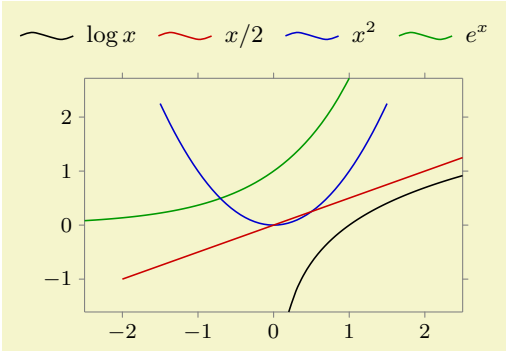
/tikz/data visualization/legend options/south west outside (no value)




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=south west outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/north outside (no value)

The legend is placed above the data. Note that the legend entries now for a row rather than a column.

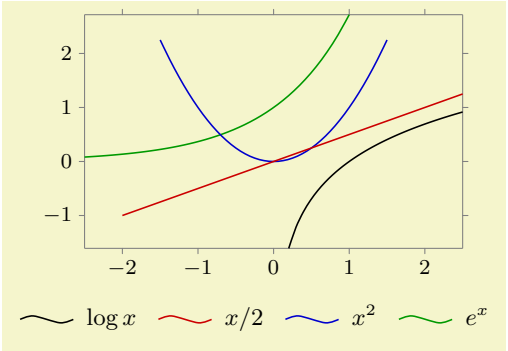



```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=north outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/above (no value)

This is an easier-to-remember alias.

/tikz/data visualization/legend options/south outside (no value)




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=south outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

/tikz/data visualization/legend options/below (no value)

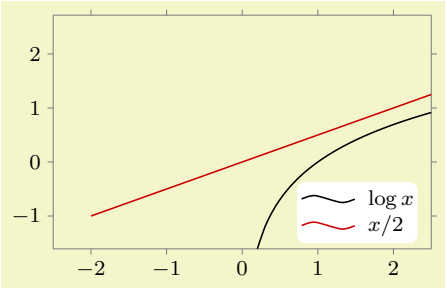
This is an easier-to-remember alias.

84.9.5 Legend Placement: Inside to the Data Visualization

There are two sets of options for placing a legend directly inside a data visualization: First, there are options for placing it inside, but next to some part of the border. Second, there are options for positioning it relative to a coordinate given by a certain data point.

/tikz/data visualization/legend options/south east inside (no value)

Puts the legend in the upper right corner of the data.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin},
  legend=south east inside,
  log=    {label in legend={text=$\log x$}},
  lin=    {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

Note that the text is now a little smaller since there tends to be much less space inside the data visualization than next to it. Also, the legend's node is filled in white by default to ensures that the legend is clearly legible even in the presence of, say, a grid or data points behind it. This behavior is triggered by the following style key:

`/tikz/data visualization/legend options/every legend inside` (style, no value)

Executed the keys `opaque` by default and sets the text size to the size of footnotes.

In order to further configure the default appearance of an inner legend, the following keys might be useful:

`/tikz/data visualization/legend options/opaque=`⟨*color*⟩ (default `white`)

When this key is used, the legend's node will be filled with the ⟨*color*⟩ and its corners will be rounded. Additionally, the inner and outer separations will be set to sensible values.

`/tikz/data visualization/legend options/transparent` (no value)

Sets the filling of the legend node to `none`.

The following keys work much the same way as `south east inside`:

`/tikz/data visualization/legend options/east inside` (no value)

`/tikz/data visualization/legend options/north east inside` (no value)

`/tikz/data visualization/legend options/south west inside` (no value)

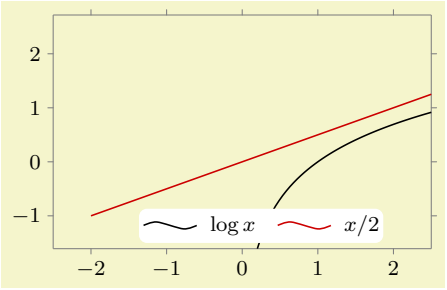
`/tikz/data visualization/legend options/west inside` (no value)

`/tikz/data visualization/legend options/north west inside` (no value)

The keys `south inside` and `north inside` are a bit different: They use a row rather than a column for the legend entries:

`/tikz/data visualization/legend options/south inside` (no value)

Puts the legend in the upper right corner of the data. Note that the text is now a little smaller since there tends to be much less space inside the data visualization than next to it.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend=south inside,
  log=    {label in legend={text=$\log x$}},
  lin=    {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

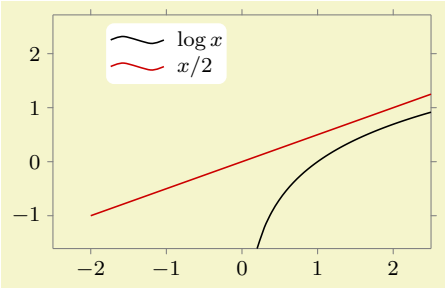
`/tikz/data visualization/legend options/north inside` (no value)

As above.

The above keys do not always give you as fine a control as you may need over the placement of the legend. In such cases, the following keys may help (or you can revert to directly setting the `at` and the `anchor` keys):

`/tikz/data visualization/legend options/at values`=⟨*data point*⟩ (no default)

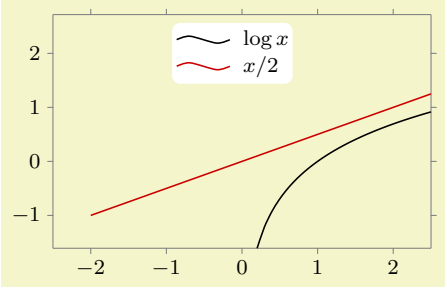
This key allows you to specify the desired center of the legend in terms of a data point. The ⟨*data point*⟩ should be a list of comma-separated key–value pairs that specify a data point. The legend will then be centered at this data point.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend={at values={x=-1, y=2}},
  log=    {label in legend={text=$\log x$}},
  lin=    {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

`/tikz/data visualization/legend options/right of`=⟨*data point*⟩ (no default)

Works like `at values`, but the anchor is set to `west`:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list={log, lin},
  legend={right of={x=-1, y=2}},
  log=    {label in legend={text=$\log x$}},
  lin=    {label in legend={text=$x/2$}},
  style sheet=strong colors]
data group {function classes};
```

The following keys work similarly:

`/tikz/data visualization/legend options/above right of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/above of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/above left of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/left of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/below left of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/below of`=⟨*data point*⟩ (no default)

`/tikz/data visualization/legend options/below right of`=⟨*data point*⟩ (no default)

84.9.6 Legend Entries: General Styling

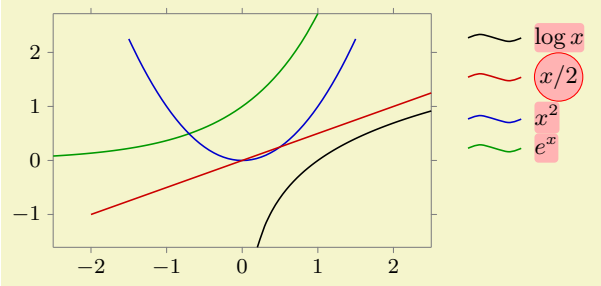
The entries in a legend can be styled in several ways:

- You can configure the styling of the text node.
- You can configure the relative placement of the text node and the little picture depicting the data set's styling.
- You can configure how the data set's styling is depicted.

Before we have look at how each of these are configured, in detail, let us first have a look at the keys that allow us to save a set of such styles:

`/tikz/data visualization/every label in legend` (style, no value)

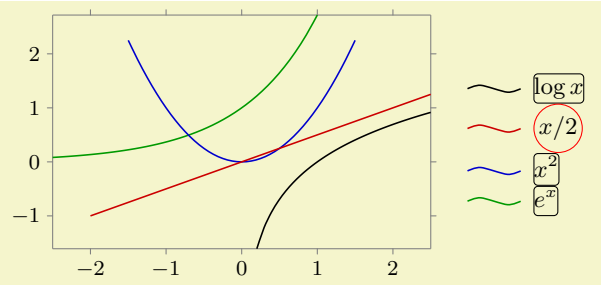
This key is executed with every label in a legend. However, the options stored in this style are executed with the path prefix `/tikz/data visualization/legend entry options`. Thus, this key can use keys like `node style` to configure the styling of all text nodes:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  every label in legend/.style={node style=
    {fill=red!30}},
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=north east outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$,
      node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

`/tikz/data visualization/legend options/label style`=⟨*options*⟩ (no default)

This key can be used with a legend. It will simply add the ⟨*options*⟩ to the `every label in legend` style for the given legend.



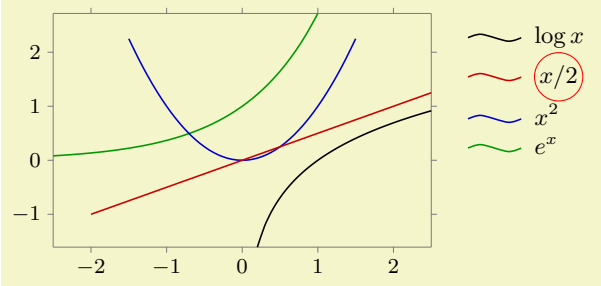

```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={label style={node style=draw}},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$,
      node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

84.9.7 Legend Entries: Styling the Text Node

The appearance of the text nodes is easy to configure.

`/tikz/data visualization/legend entry options/node style`=⟨*options*⟩ (no default)

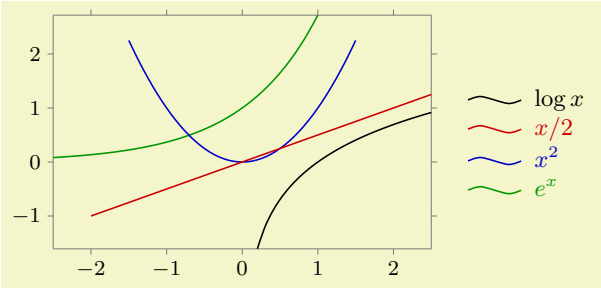
This key adds ⟨*options*⟩ to the styling of the text nodes of the label.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend=north east outside,
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$,
      node style={circle, draw=red}}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

`/tikz/data visualization/legend entry options/text colored` (no value)

Causes the `node style` to set the text color to `visualizer color`. The effect of this is that the label's text will have the same color as the data set to which it is attached.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={label style=text colored},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

84.9.8 Legend Entries: Text Placement

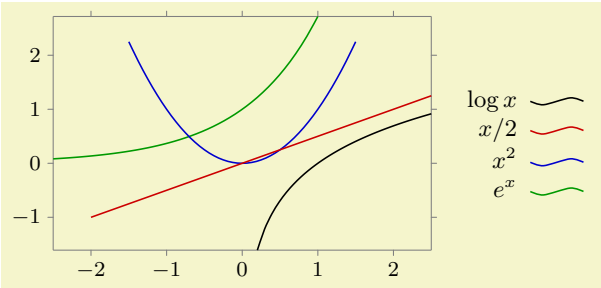
Three keys govern where the text will be placed relative to the data set style visualization.

`/tikz/data visualization/legend entry options/text right` (no value)

Placed the text node to the right of the data set style visualization. This is the default for most, but not all, legends.

`/tikz/data visualization/legend entry options/text left` (no value)

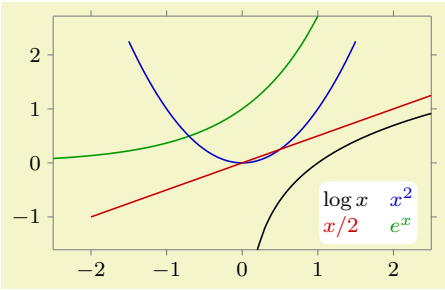
Placed the text node to the left of the data set style visualization.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={label style=text left},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

`/tikz/data visualization/legend entry options/text only` (no value)

Shows only the text nodes and no data set style visualization at all. This options only makes sense in conjunction with the `text colored` options, which is why this options is also selected implicitly.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes,
  visualize as smooth line/.list=
    {log, lin, squared, exp},
  legend={south east inside, rows=2,
          label style=text only},
  log=     {label in legend={text=$\log x$}},
  lin=     {label in legend={text=$x/2$}},
  squared={label in legend={text=$x^2$}},
  exp=     {label in legend={text=$e^x$}},
  style sheet=strong colors]
data group {function classes};
```

84.9.9 Advanced: Labels in Legends and Their Visualizers

The following explanations are important only for you if you intend to create a new visualizer and an accompanying label in legend visualizer; otherwise you can safely proceed with the next section.

A legend entry consists not only of some explaining text, but, even more importantly, of a visual representation of the style used for the data points, created by a *label in legend visualizer*. For instance, when data points are visualized as lines in different colors, the legend entry for the first line might consist of the text “first experiment” and a short line in black and the second entry might consist of “failed experiment” and a short line in red – assuming, of course, that the style sheet makes the first line black and the second

line blue. As another example, when data sets are visualized as clouds of plot marks, the texts in the legend would be accompanied by the plot marks used to visualize the data sets.

For every visualizer, the *label in legend visualizer* creates an appropriate visualization of the data set's styling. There may be more than one possible such label in legend visualizer that is appropriate, in which case options are used to choose between them.

Let us start with the key for creating a new legend entry. This key gets called for instance by `label in legend`:

`/tikz/data visualization/new legend entry=`⟨*options*⟩ (no default)

This key will add a new entry to the legend that is identified by the ⟨*options*⟩. For this, the ⟨*options*⟩ are executed once with the path prefix `/tikz/data visualization/legend entry options` and the resulting setting of the `legend` key is used to pick which legend the new entry should belong to. Then, the ⟨*options*⟩ are stored away for the time being.

Later, when the legend is created, the ⟨*options*⟩ get executed once more. This time, however, the `legend` key is no longer important. Instead, the ⟨*options*⟩ that setup keys like `text` or `visualizer in legend` now play a role.

In detail, the following happens:

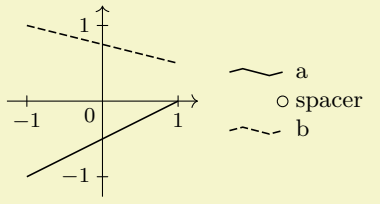
- For the legend entry, a little cell picture is created in the matrix of the legend (see Section 20.3 for details on cell pictures).
- Inside this picture, a node is created whose text is taken from the key

  ```
  /tikz/data visualization/legend entry options/text
  ```

- Also inside the picture, the code stored in the following key gets executed:

  `/tikz/data visualization/legend entry options/visualizer in legend` (no value)

  Set this key to some code that paints something in the cell picture. Typically, this will be a visual representation of the data set styling, but it could also be something different.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  style sheet=vary dashing,
  a={label in legend={text=a}},
  new legend entry={
    text=spacer,
    visualizer in legend={\draw[solid] (0,0) circle[radius=2pt];}
  },
  b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

The following styles are applied in the following order before the cell picture is filled:

1. `/tikz/data visualization/every data set label` with path `/tikz/data visualization`
2. `/tikz/data visualization/every label in legend` with path `/tikz/data visualization/legend entry options`.
3. The ⟨*options*⟩.
4. The code in the following key:

   `/tikz/data visualization/legend entry options/setup` (no value)

   Some code to be executed at this point. Mostly, it is used to setup attributes for style sheets.
5. A styling signal is emitted.
6. Only for the node: The current value of `node style`.
7. Only for the visualizer in legend: The styling that has been accumulated by calls to the following key:

   `/tikz/data visualization/legend entry options/visualizer in legend style=`
   `{`⟨*options*⟩`}` (style, no default)

   Calls to this key accumulate ⟨*options*⟩ that will be executed with the path prefix `/tikz` at this point.

As indicated earlier, the `new legend entry` key is called by the `label in legend=`⟨*options*⟩ key internally. In this case, the following extra ⟨*extra options*⟩ are passed to `new legend entry` key:

- The styling of the visualizer.
- The `/tikz/data visualization/every label in legend` style.
- The `/tikz/every label` style with path `/tikz`.
- Setting `setup` to `/data point/set=`⟨*name of the visualizer*⟩.
- The value of the `label legend options` that are stored in the visualizer. These options can be changed using the following key:

  `/tikz/data visualization/visualizer options/label in legend options=`⟨*options*⟩ (no default)

  Use this key with a visualizer to configure the label in legend options. Typically, this key is used only internally by a visualizer upon its creating to set the ⟨*options*⟩ to setup the `visualizer in legend` key.

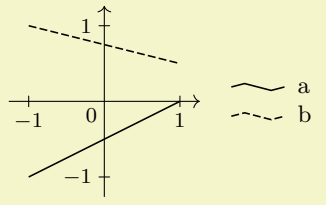
84.9.10 Reference: Label in Legend Visualizers for Lines and Scatter Plots

Visualizers like `visualize as line` or `visualize as smooth line` use a label in legend visualizer that draws a short line to represent the data set inside the legend. However, this line needs not be a simple straight line, but can be a little curve or a small circle – indeed, even the default line is not a simple straight line but rather a small zig-zag curve. To configure this line, the two keys are used, although you will only rarely use them directly, but rather use one of the predefined styles mentioned later on.

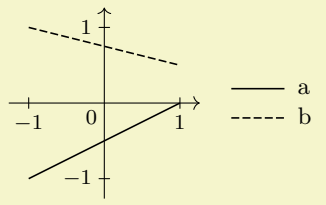
Before we go into the glorious details of all of these keys, let us first have a look at the keys you are most likely to use in practice: The keys for globally reconfiguring the default label in legend visualizers:

`/tikz/data visualization/legend entry options/default label in legend path` (style, no value)

This style is set, by default, to `zig zag label in legend line`. It is installed by the styles `straight line`, `smooth line`, and `gap line`, so changing this style will change the appearance of lines in legends. The main other sensible option for this key is `straight label in legend line`.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  style sheet=vary dashing,
  a={label in legend={text=a}},  b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```



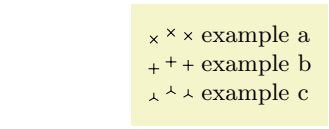

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend entry options/default label in legend path/.style=
    straight label in legend line,
  style sheet=vary dashing,
  a={label in legend={text=a}},  b={label in legend={text=b}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

`/tikz/data visualization/legend entry options/default label in legend closed path` (style, no value)

This style is executed by `smooth cycle` and `straight cycle`. There are (currently) no other predefined sets of coordinates that can be used instead of the default value `circular label in legend line`.

`/tikz/data visualization/legend entry options/default label in legend mark` (style, no value)

This style is executed by `no lines` and, implicitly, by scatter plots. The default is to use `label in legend line one mark`. Another possible value is `label in legend line three marks`.



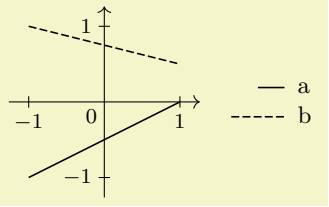

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  visualize as scatter/.list={a,b,c},
  style sheet=cross marks,
  legend entry options/default label in legend mark/.style=
    label in legend three marks,
  a={label in legend={text=example a}},
  b={label in legend={text=example b}},
  c={label in legend={text=example c}}];
```

`/tikz/data visualization/legend entry options/label in legend line coordinates=`
`{`⟨*list of coordinates*⟩`}` (no default)

This key takes a ⟨*list of coordinates*⟩, which are Ti*k*Z-coordinates separated by commas like `(0,0), (1,1)`. The effect of setting the key is the following: The label in legend visualizer used by, for instance, `visualize as line` will draw a path going through these points. When the line is drawn, the exact same style will be used as was used for the data set. For instance, if the `smooth line` key was used and also the `style=red` key, the line through the ⟨*list of coordinates*⟩ will also be red and smooth. When the `straight cycle` key was used, the coordinates will also be connected by a cycle, and so on.

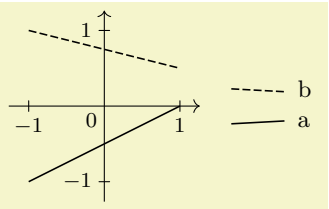
When the line connecting the ⟨*list of coordinates*⟩ is drawn, the coordinate system will have been shifted and transformed in such a way that `(0,0)` lies to the left of the text and at half the height of the character "x". This means that the right-most-point in the list should usually be `(0,0)` and all other x-coordinates should usually be negative. When the `text left` options is used, the coordinate system will have been flipped, so the ⟨*list of coordinates*⟩ is independent of whether the text is to the right or to the left of the line.

Let us now have a look at a first, simple example. We create a legend entry that is just a straight line, so it should start somewhere to the left of the origin at height 0 and go to the origin:



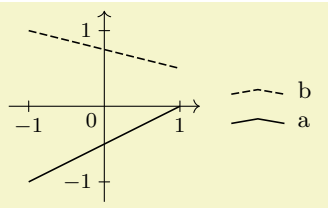

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  style sheet=vary dashing,
  a={label in legend={text=a,
      label in legend line coordinates={(-1em,0), (0,0)}}},
  b={label in legend={text=b,
      label in legend line coordinates={(-2em,0), (0,0)}}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

Now let us make this a bit more fancy and useful by using shifted lines:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend={up then right}, style sheet=vary dashing,
  a={label in legend={text=a,
      label in legend line coordinates={(-2em,-.25ex), (0,0)}}},
  b={label in legend={text=b,
      label in legend line coordinates={(-2em,.25ex), (0,0)}}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

In the final example, we use a little "hat" to represent lines:

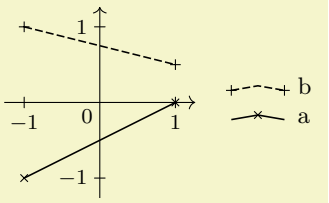



```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend={up then right}, style sheet=vary dashing,
  a={label in legend={text=a,
      label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}},
  b={label in legend={text=b,
      label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)}}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

`/tikz/data visualization/legend entry options/label in legend mark coordinates=`
{⟨*list of coordinates*⟩} (no default)

This key is similar to `label in legend line coordinates`, but now the ⟨*list of coordinates*⟩ is used as the positions where plot marks are shown. Naturally, plot marks are only shown there if they are also shown by the visualizer in the actual data – just like the line through the coordinates of the previous key is only shown when there is a line.

The ⟨*list of coordinates*⟩ may be the same as the one used for lines, but usually it is not. In general, it is better to have marks for instance not at the ends of the line.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  school book axes, visualize as line/.list={a,b},
  legend={up then right},
  style sheet=vary dashing,
  style sheet=cross marks,
  a={label in legend={text=a,
      label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)},
      label in legend mark coordinates={
        (-1em,.2ex)}}},
  b={label in legend={text=b,
      label in legend line coordinates={
        (-2em,-.2ex), (-1em,.2ex), (0,-.2ex)},
      label in legend mark coordinates={
        (-2em,-.2ex), (0,-.2ex)}}}]
data point [x=-1, y=-1, set=a]   data point [x=1, y=0, set=a]
data point [x=-1, y=1,  set=b]   data point [x=1, y=0.5, set=b];
```

Naturally, you typically will not give coordinates explicitly for each label, but use one of the following styles:

`/tikz/data visualization/legend entry options/straight label in legend line` (no value)

Just gives a straight line and two plot marks.

example

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [visualize as line,
  line={style={mark=x}, label in legend={text=example,
    straight label in legend line}}];
```

This style might seem like a good idea to use in general, but it does have a huge drawback: Some commonly used plot marks will be impossible to distinguish – even though there is no problem distinguishing them in a graph.

bad example a
bad example b
bad example c

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [visualize as line/.list={a,b,c},
  legend entry options/default label in legend path/.style=
    straight label in legend line,
  a={style={mark=+}, label in legend={text=bad example a}},
  b={style={mark=-}, label in legend={text=bad example b}},
  c={style={mark=|}, label in legend={text=bad example c}}];
```

For this reason, this option is not the default, but rather the next one.

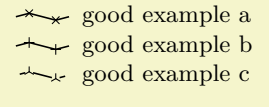
`/tikz/data visualization/legend entry options/zig zag label in legend line` (no value)

Uses a small up-down-up line as the label in legend visualizer. The two plot marks are at the extremal points of the line. It works pretty well in almost all situations and is the default.

better example a
better example b
better example c

```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  visualize as line=a,
  visualize as smooth line/.list={b,c},
  a={style={mark=+}, label in legend={text=better example a}},
  b={style={mark=-}, label in legend={text=better example b}},
  c={style={mark=|}, label in legend={text=better example c}}];
```

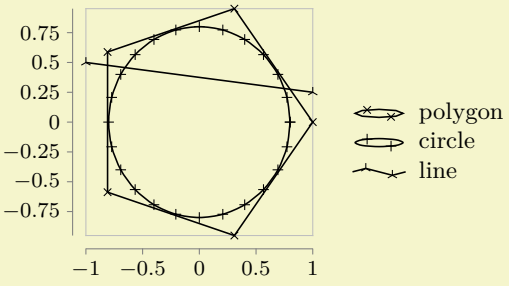
Even though the above example shows that the marks are easier to distinguish than with a straight line, the chosen marks are still not optimal. This is the reason that the `cross marks` style uses different crosses:




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [
  visualize as line/.list={a,b},
  visualize as smooth line=c,
  style sheet=cross marks,
  a={label in legend={text=good example a}},
  b={label in legend={text=good example b}},
  c={gap line, label in legend={text=good example c}}];
```

`/tikz/data visualization/legend entry options/circular label in legend line` (no value)

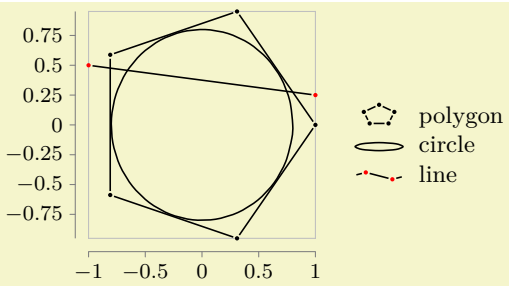
This style is especially tailored to represent lines that are closed. It is automatically selected for instance by the `polygon` or the `smooth cycle` styles.




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c},
  a={polygon}, b={smooth cycle},
  style sheet=cross marks,
  a={label in legend={text=polygon}},
  b={label in legend={text=circle}},
  c={label in legend={text=line}}]
data [format=function, set=a] {
  var t : {0,72,...,359};
  func x = cos(\value t);
  func y = sin(\value t);
}
data [format=function, set=b] {
  var t : [0:2*pi];
  func x = .8*cos(\value t r);
  func y = .8*sin(\value t r);
}
data point [x=-1, y=0.5, set=c]
data point [x=1,  y=0.25, set=c];
```

`/tikz/data visualization/legend entry options/gap circular label in legend line` (no value)

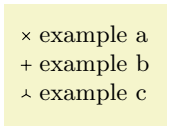
This style is especially tailored to for the `gap cycle` style and automatically selected by it:




```
\usetikzlibrary {datavisualization.formats.functions}
\tikz \datavisualization [
  scientific axes={clean}, all axes={length=3cm},
  visualize as line/.list={a,b,c},
  a={gap cycle}, b={smooth cycle}, c={gap line},
  a={style={mark=*, mark size=0.5pt},
     label in legend={text=polygon}},
  b={label in legend={text=circle}},
  c={style={mark=*, mark size=0.5pt, mark options=red},
     label in legend={text=line}}]
data [format=function, set=a] {
  var t : {0,72,...,359};
  func x = cos(\value t);
  func y = sin(\value t);
}
data [format=function, set=b] {
  var t : [0:352];
  func x = .8*cos(\value t);
  func y = .8*sin(\value t);
}
data point [x=-1, y=0.5, set=c]
data point [x=1,  y=0.25, set=c];
```

`/tikz/data visualization/legend entry options/label in legend one mark` (no value)

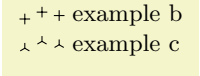
To be used with scatter plots, since no line is drawn. Just displays a single mark (this is the default with a scatter plot or when the `no line` is selected.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [visualize as scatter/.list={a,b,c},
   style sheet=cross marks,
  a={label in legend={text=example a}},
  b={label in legend={text=example b}},
  c={label in legend={text=example c}}];
```

`/tikz/data visualization/legend entry options/label in legend three marks` (no value)

An alternative to the previous style, where several marks are shown.




```
\usetikzlibrary {datavisualization}
\tikz \datavisualization [visualize as scatter/.list={a,b,c},
  style sheet=cross marks,
  a={label in legend={text=example a, label in legend three marks}},
  b={label in legend={text=example b, label in legend three marks}},
  c={label in legend={text=example c, label in legend three marks}}];
```

85 Polar Axes

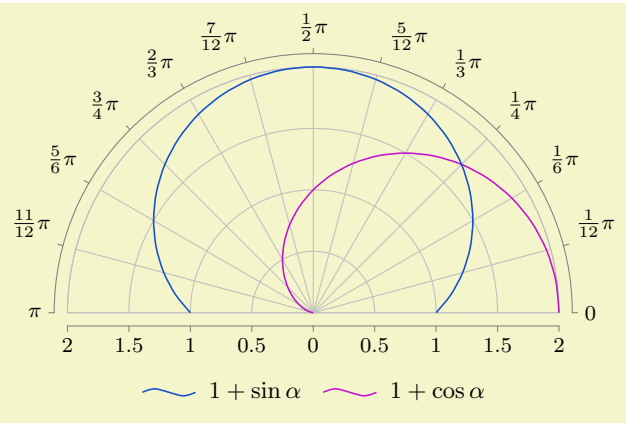
85.1 Overview

TikZ Library datavisualization.polar

```
\usetikzlibrary{datavisualization.polar} % LaTeX and plain TeX
\usetikzlibrary[datavisualization.polar] % ConTeXt
```

This library contains keys that allow you to create plots in a polar axis system is used.

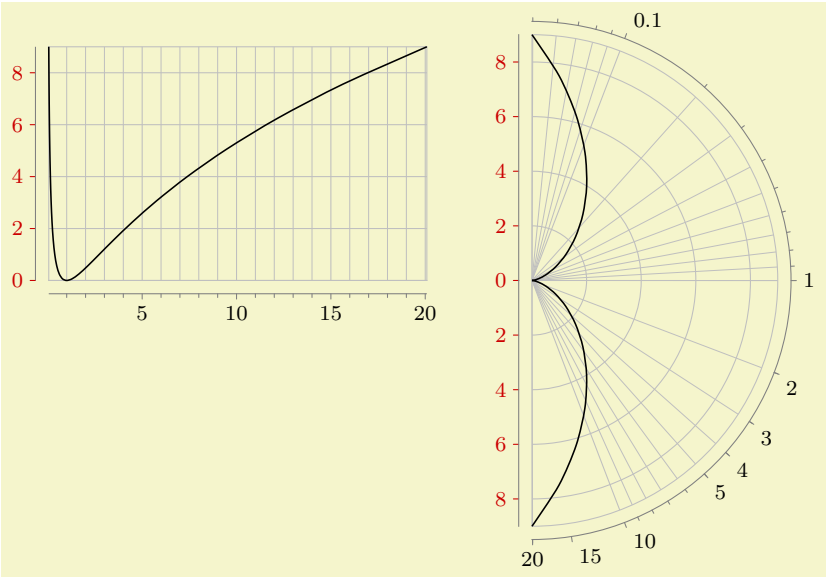
In a *polar axis system* two attributes are visualized by displacing a data point as follows: One attribute is used to compute a an angle (a direction) while a second attribute is used as a radius (a distance). The angle can be measured in degrees, radians, or can be scaled arbitrarily.




```
\usetikzlibrary { datavisualization.formats.functions,
datavisualization.polar, }
\tikz \datavisualization [
  scientific polar axes={0 to pi, clean},
  all axes=grid,
  style sheet=vary hue,
  legend=below
  ]
  [visualize as smooth line=sin,
   sin={label in legend={text=$1+\sin \alpha$}}]
  data [format=function] {
    var  angle : interval [0:pi];
    func radius = sin(\value{angle}r) + 1;
  }
  [visualize as smooth line=cos,
   cos={label in legend={text=$1+\cos\alpha$}}]
  data [format=function] {
    var  angle : interval [0:pi];
    func radius = cos(\value{angle}r) + 1;
  };
```

Most of the time, in order to create a polar axis system, you will just use the `scientific polar axes` key, which takes a number of options that allow you to configure the axis system in greater detail. This key is documented in Section 85.2. Internally, this key uses more low level keys which are documented in the en suite sections.

It is worthwhile to note that the axes of a polar axis system are, still, normal axes of the data visualization system. In particular, all the configurations possible for, say, Cartesian axes also apply to the "angle axis" and the "radius axis" of a polar axis system. For instance, you can could make both axes logarithmic or style their ticks:



```
\usetikzlibrary { datavisualization.formats.functions, datavisualization.polar, }
\tikz[baseline] \datavisualization [
  scientific axes=clean,
  x axis={attribute=angle, ticks={minor steps between steps=4}},
  y axis={attribute=radius, ticks={some, style=red!80!black}},
  all axes=grid,
  visualize as smooth line=sin]
  data [format=function] {
    var t : interval [-3:3];
    func angle = exp(\value t);
    func radius = \value{t}*\value{t};
  };
\qquad
\tikz[baseline] \datavisualization [
  scientific polar axes={right half clockwise, clean},
  angle axis={logarithmic,
    ticks={
      minor steps between steps=8,
      major also at/.list={2,3,4,5,15,20}}},
  radius axis={ticks={some, style=red!80!black}},
  all axes=grid,
  visualize as smooth line=sin]
  data [format=function] {
    var t : interval [-3:3];
    func angle = exp(\value t);
    func radius = \value{t}*\value{t};
  };
```

85.2 Scientific Polar Axis System

`/tikz/data visualization/scientific polar axes=`⟨*options*⟩ (no default)

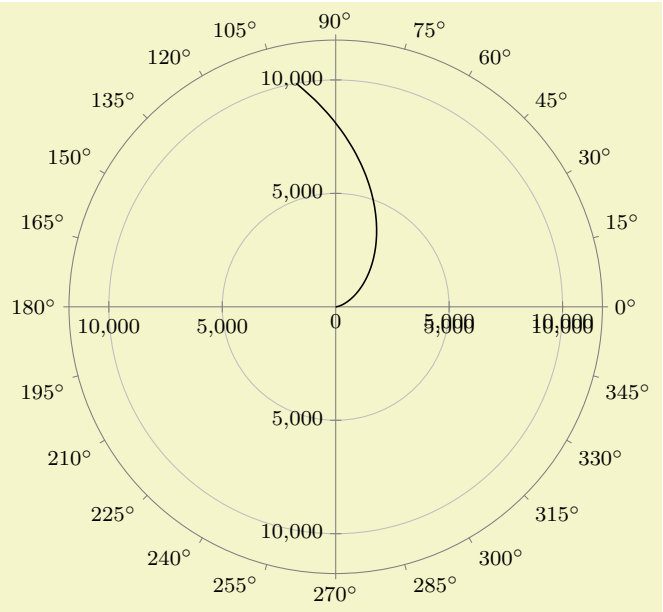
This key installs a polar axis system that can be used in a "scientific" publication. Two axes are created called the `angle axis` and the `radius axis`. Unlike "normal" Cartesian axes, these axes do not point in a specific direction. Rather, the `radius axis` is used to map the values of one attribute to a distance from the origin while the `angle axis` is used to map the values of another attribute to a rotation angle.

The ⟨*options*⟩ will be executed with the path prefix

```
/tikz/data visualization/scientific polar axes
```

The permissible keys are documented in the later subsections of this section.

Let us start with the configuration of the radius axis since it is easier. Firstly, you should specify which attribute is linked to the radius. The default is `radius`, but you will typically wish to change this. As with any other axis, the `attribute` key is used to configure the axis, see Section 82.2.2 for details. You can also apply all other configurations to the radius axis like, say, `unit length` or `length` or `style`. Note, however, that the `logarithmic` key will not work with the radius axis for a `scientific polar axes` system since the attribute value zero is always placed at the center – and for a logarithmic plot the value `0` cannot be mapped.



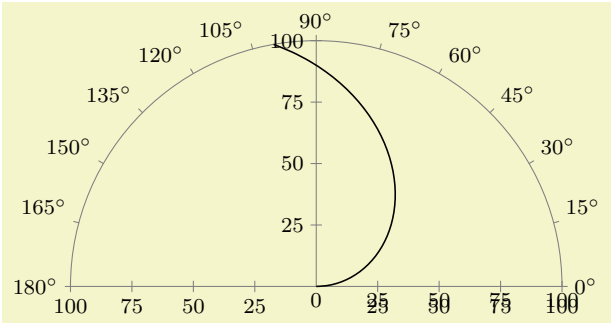

```
\usetikzlibrary { datavisualization.formats.functions,
datavisualization.polar, }
\tikz \datavisualization [
  scientific polar axes,
  radius axis={
    attribute=distance,
    ticks={step=5000},
    padding=1.5em,
    length=3cm,
    grid
  },
  visualize as smooth line]
data [format=function] {
  var  angle : interval [0:100];
  func distance = \value{angle}*\value{angle};
};
```

For the `angle axis`, you can also specify an attribute using the `attribute` key. However, for this axis the mapping of a value to an actual angle is a complicated process involving many considerations of how the polar axis system should be visualized. For this reason, there are a large number of predefined such mappings documented in Section 85.2.2. Finally, as for a `scientific plot`, you can configure where the ticks should be shown using the keys `inner ticks`, `outer ticks`, and `clean`, documented below.

85.2.1 Tick Placements

`/tikz/data visualization/scientific polar axes/outer ticks` (no value)

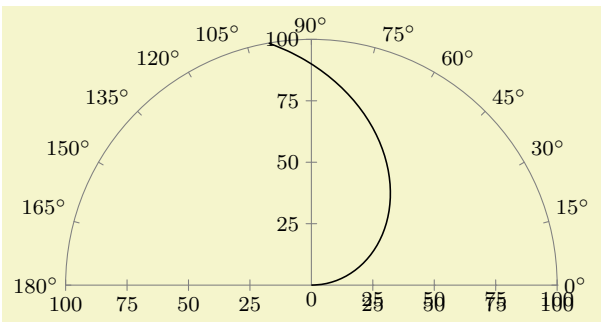
This key, which is the default, causes ticks to be drawn "outside" the outer "ring" of the polar axes:




```
\usetikzlibrary { datavisualization.formats.functions,
datavisualization.polar, }
\tikz \datavisualization [
  scientific polar axes=outer ticks, 0 to 180,
  visualize as smooth line]
data [format=function] {
  var  angle : interval [0:100];
  func radius = \value{angle};
};
```

`/tikz/data visualization/scientific polar axes/inner ticks` (no value)

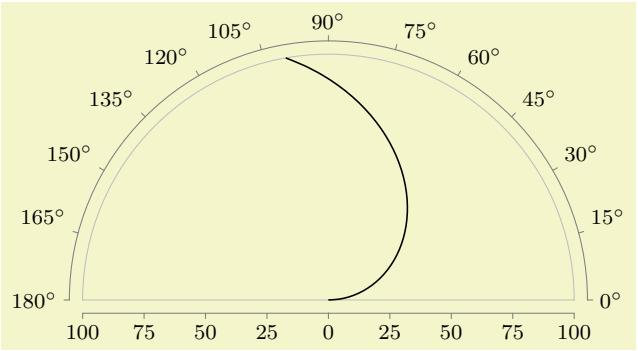
This key causes the ticks to be "turned to the inside". I do not recommend using this key.




```
\usetikzlibrary { datavisualization.formats.functions,
datavisualization.polar, }
\tikz \datavisualization [
  scientific polar axes=inner ticks, 0 to 180,
  visualize as smooth line]
data [format=function] {
  var  angle : interval [0:100];
  func radius = \value{angle};
};
```

/tikz/data visualization/scientific polar axes/clean (no value)

This key separates the area where the data is shown from the area where the ticks are shown. Usually, this is the best choice for the tick placement since it avoids a collision of data and explanations.




```
\usetikzlibrary { datavisualization.formats.functions,
datavisualization.polar, }
\tikz \datavisualization [
  scientific polar axes={clean, 0 to 180},
  visualize as smooth line]
data [format=function] {
  var  angle : interval [0:100];
  func radius = \value{angle};
};
```

85.2.2 Angle Ranges

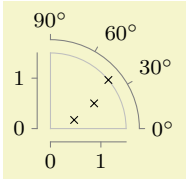
Suppose you create a polar plot in which the radius values vary between, say, 567 and 1234. Then the normal axis scaling mechanisms can be used to compute a good scaling for the "radius axis": Place the value 1234 at a distance of , say, 5 cm from the origin and place the value 0 at the origin. Now, by comparison, suppose that the values of the angle axis's attribute ranged between, say, 10 and 75.7. In this case, we may wish the angles to be scaled so that the minimum value is horizontal and the maximum value is vertical. But we may also wish the a value of 0 is horizontal and a value of 90 is vertical.

Since it is unclear which interpretation is the right one, you have to use an option to select which should happen. The applicable options fall into three categories:

- Options that request the scaling to be done in such a way that the attribute is interpreted as a value in degrees and such that the minimum and maximum of the depicted range is a multiple of 90°. For instance, the option `0 to 180` causes the angle axis to range from 0° to 180°, independently of the actual range of the values.
- Options that work as above, but use radians rather than degrees. An example is the option `0 to pi`.
- Options that map the minimum value in the data to a horizontal or vertical line and the maximum value to another such line. This is useful when the values neither directly correspond to degrees or radians. In this case, the angle axis may also be a logarithmic axis.

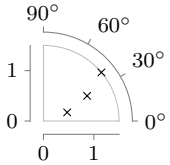
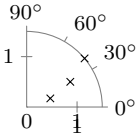
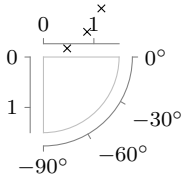
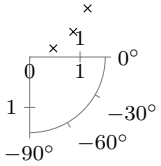
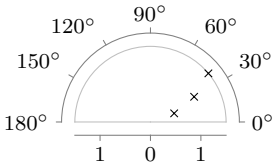
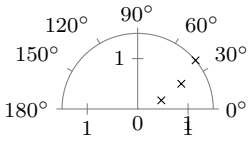
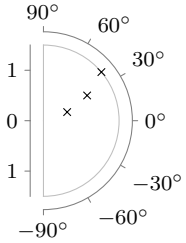
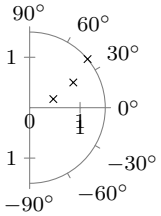
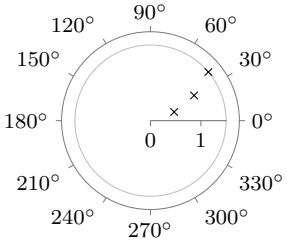
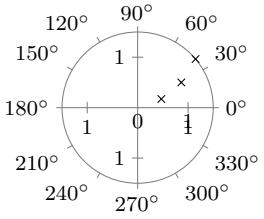
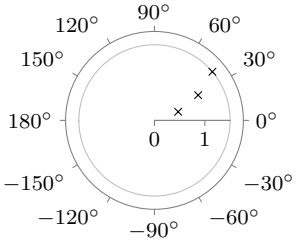
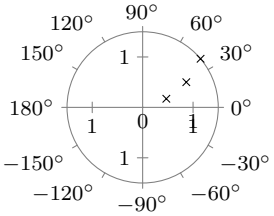
In addition to the above categories, all of the option documented in the following implicitly also select quadrants that are used to depict the data. For instance, the `0 to 90` key and also the `0 to pi half` key setup the polar axis system in such a way that only first (upper right) quadrant is used. No check is done whether the data fill actually lie in this quadrant – if it does not, the data will "bleed outside" the range. Naturally, with a key like `0 to 360` or `0 to 2pi` this cannot happen.

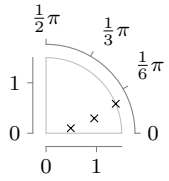
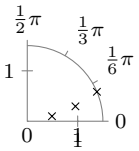
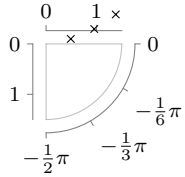
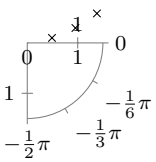
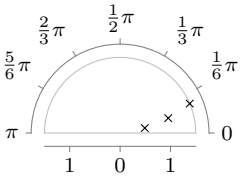
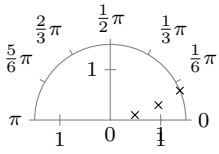
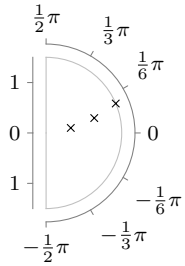
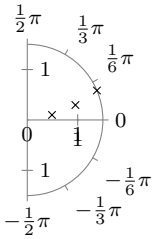
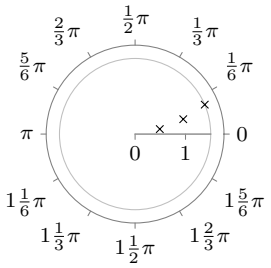
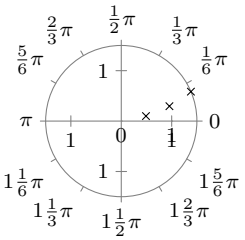
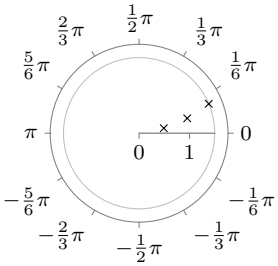
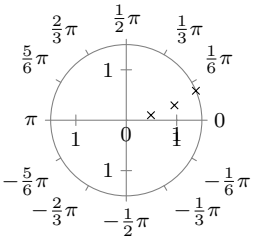
In order to save some space in this manual, in the following the different possible keys are only given in a table together with a small example for each key. The examples were created using the following code:

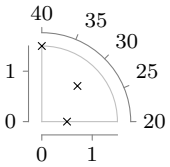
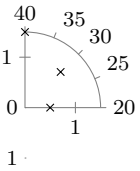
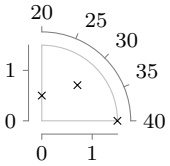
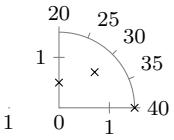
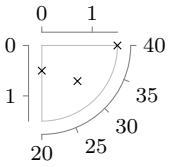
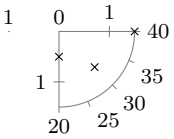
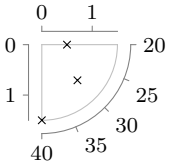
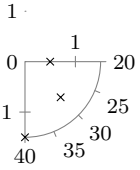
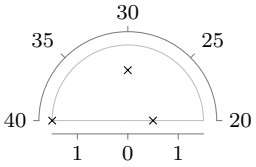
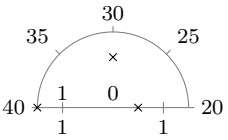
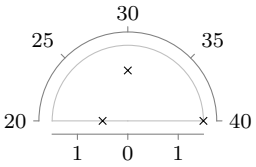
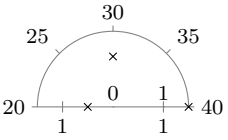
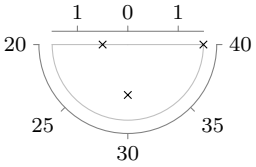
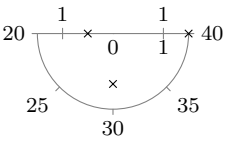
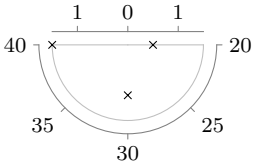
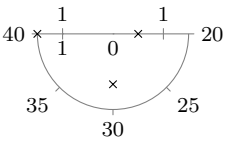


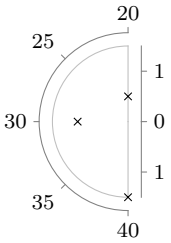
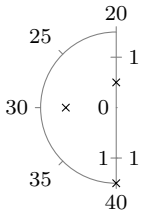
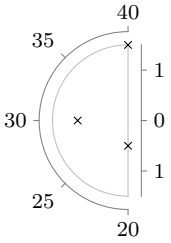
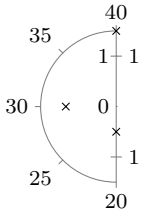
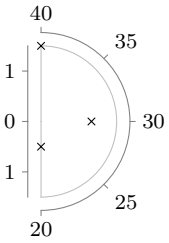
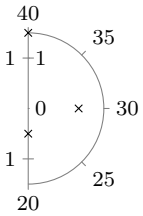
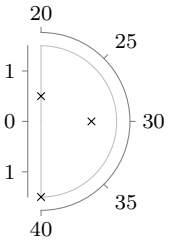
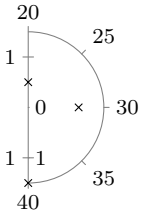

```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization [
  scientific polar axes={
    clean,
    0 to 90  % the option
  },
  angle axis={ticks={step=30}},
  radius axis={length=1cm, ticks={step=1}},
  visualize as scatter]
data point [angle=20, radius=0.5]
data point [angle=30, radius=1]
data point [angle=40, radius=1.5];
```

For the options on radians, the angle values have been replaced by `0.2`, `0.3`, and `0.4` and the stepping has been changed by setting `step=(pi/6)`. For the quadrant options, no stepping is set at all (it is computed automatically).

Option	*With clean ticks*	*With outer ticks*
`0 to 90`		
`-90 to 0`		
`0 to 180`		
`-90 to 90`		
`0 to 360`		
`-180 to 180`		

Option	*With clean ticks*	*With outer ticks*
`0 to pi half`		
`-pi half to 0`		
`0 to pi`		
`-pi half to pi half`		
`0 to 2pi`		
`-pi to pi`		

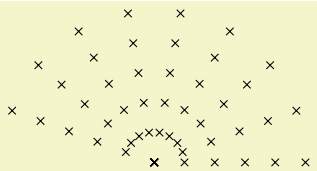
Option	*With clean ticks*	*With outer ticks*
`quadrant`		
`quadrant clockwise`		
`fourth quadrant`		
`fourth quadrant clockwise`		
`upper half`		
`upper half clockwise`		
`lower half`		
`lower half clockwise`		

Option	*With clean ticks*	*With outer ticks*
`left half`		
`left half clockwise`		
`right half`		
`right half clockwise`		

85.3 Advanced: Creating a New Polar Axis System

`/tikz/data visualization/new polar axes={`⟨*angle axis name*⟩`}{`⟨*radius axis name*⟩`}` (no default)

This key actually creates two axes, whose names are give as parameters: An *angle axis* and a *radius axis*. These two axes work in concert in the following way: Suppose a data point has two attributes called `angle` and `radius` (these attribute names can be changed by changing the `attribute` of the ⟨*angle axis name*⟩ or the ⟨*radius axis name*⟩, respectively). These two attributes are then scaled as usual, resulting in two "reasonable" values a (for the angle) and r (for the radius). Then, the data point gets visualized (in principle, details will follow) at a position on the page that is at a distance of r from the origin and at an angle of a.



```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes={angle axis}{radius axis},
     radius axis={length=2cm},
     visualize as scatter]
  data [format=named] {
    angle={0,20,...,160}, radius={0,...,5}
  };
```

In detail, the ⟨*angle axis*⟩ keeps track of two vectors v_0 and v_{90}, each of which will usually have unit length (length `1pt`) and which point in two different directions. Given a radius r (measured in TEX `pt`s, so if the radius attribute `10pt`, then r would be 10) and an angle a, let s be the sine of a and let c be the cosine of a, where a is a number is degrees (so s would be 1 for $a = 90$). Then, the current page position is shifted by $c \cdot r$ times v_0 and, additionally, by $s \cdot r$ times v_{90}. This means that in the "polar

coordinate system" v_0 is the unit vector along the "0°-axis" and v_{90} is the unit vector along "90°-axis". The values of v_0 and v_{90} can be changed using the following key on the ⟨*angle axis*⟩:

`/tikz/data visualization/axis options/unit vectors={`⟨*unit vector 0 degrees*⟩`}{`⟨*unit vector 90 degrees*⟩`}` (no default, initially `{(1pt,0pt)}{(0pt,1pt)}`)

Both the ⟨*unit vector 0 degrees*⟩ and the ⟨*unit vector 90 degrees*⟩ are TikZ coordinates:



```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes={angle axis}{radius axis},
     radius axis={unit length=1cm},
     angle axis={unit vectors={(10:1pt)}{(60:1pt)}},
     visualize as scatter]
  data [format=named] {
    angle={0,90}, radius={0.25,0.5,...,2}
  };
```

Once created, the `angle axis` can be scaled conveniently using the following keys:

`/tikz/data visualization/axis options/degrees` (no value)

When this key is passed to the angle axis of a polar axis system, it sets up the scaling so that a value of `360` on this axis corresponds to a complete circle.



```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes={angle axis}{radius axis},
     radius axis={unit length=1cm},
     angle axis={degrees},
     visualize as scatter]
  data [format=named] {
    angle={10,90}, radius={0.25,0.5,...,2}
  };
```

`/tikz/data visualization/axis options/radians` (no value)

In contrast to `degrees`, this option sets up things so that a value of `2*pi` on this axis corresponds to a complete circle.



```
\usetikzlibrary {datavisualization.polar}
\tikz \datavisualization
    [new polar axes={angle axis}{radius axis},
     radius axis={unit length=1cm},
     angle axis={radians},
     visualize as scatter]
  data [format=named] {
    angle={0,1.5}, radius={0.25,0.5,...,2}
  };
```

86 The Data Visualization Backend

86.1 Overview

The present section explains the mechanisms behind the data visualization engine.

Until it is documented properly, we will have to make do with the documentation in the source code.

86.2 The Rendering Pipeline

To be written...

86.3 Usage

To be written...

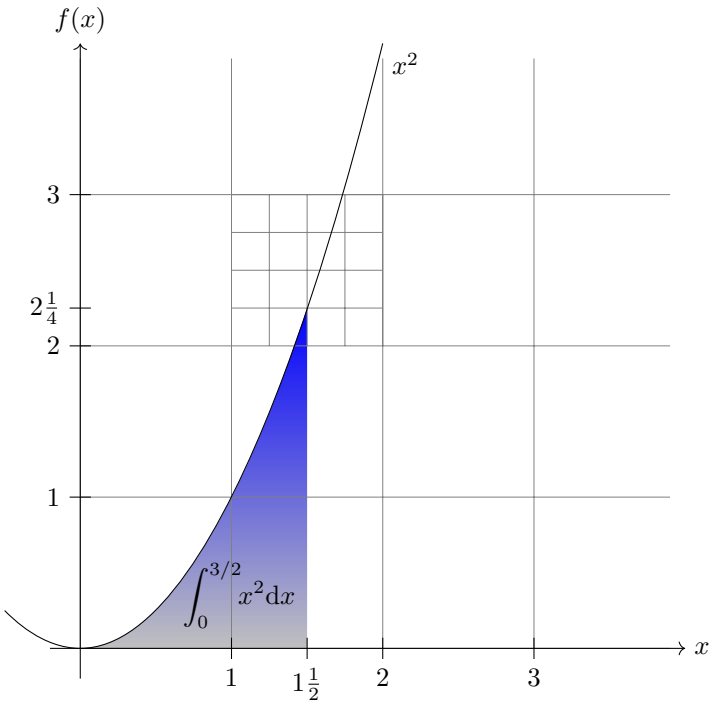
86.4 The Mathematical Micro-Kernel

To be written...

Part VII
Utilities

by Till Tantau

The utility packages are not directly involved in creating graphics, but you may find them useful nonetheless. All of them either directly depend on PGF or they are designed to work well together with PGF even though they can be used in a stand-alone way.




```
\begin{tikzpicture}[scale=2]
  \shade[top color=blue,bottom color=gray!50] (0,0) parabola (1.5,2.25) |- (0,0);
  \draw (1.05cm,2pt) node[above] {$\displaystyle\int_0^{3/2} \!\!x^2\mathrm{d}x$};

  \draw[help lines] (0,0) grid (3.9,3.9)
       [step=0.25cm]      (1,2) grid +(1,1);

  \draw[->] (-0.2,0) -- (4,0) node[right] {$x$};
  \draw[->] (0,-0.2) -- (0,4) node[above] {$f(x)$};

  \foreach \x/\xtext in {1/1, 1.5/1\frac{1}{2}, 2/2, 3/3}
    \draw[shift={(\x,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\xtext$};

  \foreach \y/\ytext in {1/1, 2/2, 2.25/2\frac{1}{4}, 3/3}
    \draw[shift={(0,\y)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\ytext$};

  \draw (-.5,.25) parabola bend (0,0) (2,4) node[below right] {$x^2$};
\end{tikzpicture}
```

87 Key Management

This section describes the package `pgfkeys`. It is loaded automatically by both PGF and Ti*k*Z.

```
\usepackage{pgfkeys} % LaTeX
\input{pgfkeys.tex} % plain TeX
\usemodule[pgfkeys]  % ConTeXt
```

This package can be used independently of PGF. Note that no other package of PGF needs to be loaded (so neither the emulation layer nor the system layer is needed). The ConTeXt abbreviation is `pgfkey` if `pgfmod` is not loaded.

87.1 Introduction

87.1.1 Comparison to Other Packages

The `pgfkeys` package defines a key–value management system that is in some sense similar to the more light-weight `keyval` system and the improved `xkeyval` system. However, `pgfkeys` uses a slightly different philosophy than these systems and it will coexist peacefully with both of them.

The main differences between `pgfkeys` and `xkeyval` are the following:

- `pgfkeys` organizes keys in a tree, while `keyval` and `xkeyval` use families. In `pgfkeys` the families correspond to the root entries of the key tree.
- `pgfkeys` has no save-stack impact (you will have to read the TeXBook very carefully to appreciate this).
- `pgfkeys` is slightly slower than `keyval`, but not much.
- `pgfkeys` supports styles. This means that keys can just stand for other keys (which can stand for other keys in turn or which can also just execute some code). Ti*k*Z uses this mechanism heavily.
- `pgfkeys` supports multi-argument key code. This can, however, be emulated in `keyval`.
- `pgfkeys` supports handlers. These are call-backs that are called when a key is not known. They are very flexible, in fact even defining keys in different ways is handled by, well, handlers.

87.1.2 Quick Guide to Using the Key Mechanism

The following quick guide to PGF's key mechanism only treats the most commonly used features. For an in-depth discussion of what is going on, please consult the remainder of this section.

Keys are organized in a large tree that is reminiscent of the Unix file tree. A typical key might be, say, `/tikz/coordinate system/x` or just `/x`. Again as in Unix, when you specify keys you can provide the complete path of the key, but you usually just provide the name of the key (corresponding to the file name without any path) and the path is added automatically.

Typically (but not necessarily) some code is associated with a key. To execute this code, you use the `\pgfkeys` command. This command takes a list of so-called key–value pairs. Each pair is of the form ⟨*key*⟩=⟨*value*⟩. For each pair the `\pgfkeys` command will execute the code stored for the ⟨*key*⟩ with its parameter set to ⟨*value*⟩.

Here is a typical example of how the `\pgfkeys` command is used:

```
\pgfkeys{/my key=hallo,/your keys/main key=something\strange,
         key name without path=something else}
```

Now, to set the code that is stored in a key you do not need to learn a new command. Rather, the `\pgfkeys` command can also be used to set the code of a key. This is done using so-called *handlers*. They look like keys whose names look like "hidden files in Unix" since they start with a dot. The handler for setting the code of a key is appropriately called `/.code` and it is used as follows:

The value is 'hi!'.

```
\pgfkeys{/my key/.code=The value is '#1'.}
\pgfkeys{/my key=hi!}
```

As you can see, in the first line we defined the code for the key `/my key`. In the second line we executed this code with the parameter set to `hi!`.

There are numerous handlers for defining a key. For instance, we can also define a key whose value actually consists of more than one parameter.

The values are 'a1' and 'a2'.

```
\pgfkeys{/my key/.code 2 args=The values are '#1' and '#2'.}
\pgfkeys{/my key={a1}{a2}}
```

We often want to have keys where the code is called with some default value if the user does not provide a value. Not surprisingly, this is also done using a handler, this time called `/.default`.

(hallo)(hello)

```
\pgfkeys{/my key/.code=(#1)}
\pgfkeys{/my key/.default=hello}
\pgfkeys{/my key=hallo,/my key}
```

The other way round, it is also possible to specify that a value *must* be specified, using a handler called `/.value required`. Finally, you can also require that no value *may* be specified using `/.value forbidden`.

All keys for a package like, say, Ti*k*Z start with the path `/tikz`. We obviously do not like to write this path down every time we use a key (so we do not have to write things like `\draw[/tikz/line width=1cm]`). What we need is to somehow "change the default path to a specific location". This is done using the handler `/.cd` (for "change directory"). Once this handler has been used on a key, all subsequent keys *in the current call of* `\pgfkeys` *only* are automatically prefixed with this path, if necessary.

Here is an example:

```
\pgfkeys{/tikz/.cd,line width=1cm,line cap=round}
```

This makes it easy to define commands like `\tikzset`, which could be defined as follows (the actual definition is a bit faster, but the effect is the same):

```
\def\tikzset#1{\pgfkeys{/tikz/.cd,#1}}
```

When a key is handled, instead of executing some code, the key can also cause further keys to be executed. Such keys will be called *styles*. A style is, in essence, just a key list that should be executed whenever the style is executed. Here is an example:

(a:foo)(b:bar)(a:wow)

```
\pgfkeys{/a/.code=(a:#1)}
\pgfkeys{/b/.code=(b:#1)}
\pgfkeys{/my style/.style={/a=foo,/b=bar,/a=#1}}
\pgfkeys{/my style=wow}
```

As the above example shows, styles can also be parameterized, just like the normal code keys.

As a typical use of styles, suppose we wish to set up the key `/tikz` so that it will change the default path to `/tikz`. This can be achieved as follows:

```
\pgfkeys{/tikz/.style=/tikz/.cd}
\pgfkeys{tikz,line width=1cm,draw=red}
```

Note that when `\pgfkeys` is executed, the default path is set to `/`. This means that the first `tikz` will be completed to `/tikz`. Then `/tikz` is a style and, thus, replaced by `/tikz/.cd`, which changes the default path to `/tikz`. Thus, the `line width` is correctly prefixed with `/tikz`.

87.2 The Key Tree

The `pgfkeys` package organizes keys in a so-called *key tree*. This tree will be familiar to anyone who has used a Unix operating system: A key is addressed by a path, which consists of different parts separated by slashes. A typical key might be `/tikz/line width` or just `/tikz` or something more complicated like `/tikz/cs/x/.store in`.

Let us fix some further terminology: Given a key like `/a/b/c`, we call the part leading up the last slash (`/a/b`) the *path* of the key. We call everything after the last slash (`c`) the *name* of the key (in a file system this would be the file name).

We do not always wish to specify keys completely. Instead, we usually specify only part of a key (typically only the name) and the *default path* is then added to the key at the front. So, when the default path is `/tikz` and you refer to the (partial) key `line width`, the actual key that is used is `/tikz/line width`. There is a simple rule for deciding whether a key is a partial key or a full key: If it starts with a slash, then it is a full key and it is not modified; if it does not start with a slash, then the default path is automatically prefixed.

Remark: The above rule is actually a definition, hence the corresponding sufficiency conditions hold. That is, if it is a full key, then it starts with a slash; if it is a partial key, then it does not start with a slash. Moreover, a path always starts with a slash.

Note that the default path is not the same as a search path. In particular, the default path is just a single path. When a partial key is given, only this single default path is prefixed; `pgfkeys` does not try to look up the key in different parts of a search path. It is, however, possible to emulate search paths, but a much more complicated mechanism must be used.

When you set keys (to be explained in a moment), you can freely mix partial and full keys and you can change the default path. This makes it possible to temporarily use keys from another part of the key tree (this turns out to be a very useful feature).

Each key (may) store some *tokens* and there exist commands, described below, for setting, getting, and changing the tokens stored in a key. However, you will only very seldom use these commands directly. Rather, the standard way of using keys is the `\pgfkeys` command or some command that uses it internally like, say, `\tikzset`. So, you may wish to skip the following commands and continue with the next subsection.

`\pgfkeyssetvalue{`⟨*full key*⟩`}{`⟨*token text*⟩`}`

Stores the ⟨*token text*⟩ in the ⟨*full key*⟩. The ⟨*full key*⟩ may not be a partial key, so no default-path-adding is done. The ⟨*token text*⟩ can be arbitrary tokens and may even contain things like `#` or unbalanced TEX-ifs.

Hello, world!

```
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysvalueof{/my family/my key}
```

The setting of a key is always local to the current TEX group.

`\pgfkeyssetevalue{`⟨*full key*⟩`}{`⟨*token text*⟩`}`

The `\edef` version of `\pgfkeyssetvalue`.

`\pgfkeyslet{`⟨*full key*⟩`}{`⟨*macro*⟩`}`

Performs a `\let` statement so the ⟨*full key*⟩ points to the contents of ⟨*macro*⟩.

Hello, world!

```
\def\helloworld{Hello, world!}
\pgfkeyslet{/my family/my key}{\helloworld}
\pgfkeysvalueof{/my family/my key}
```

You should never let a key be equal to `\relax`. Such a key may or may not be indistinguishable from an undefined key.

`\pgfkeysgetvalue{`⟨*full key*⟩`}{`⟨*macro*⟩`}`

Retrieves the tokens stored in the ⟨*full key*⟩ and lets ⟨*macro*⟩ be equal to these tokens. If the key has not been set, the ⟨*macro*⟩ will be equal to `\relax`.

Hello, world!

```
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysgetvalue{/my family/my key}{\helloworld}
\helloworld
```

`\pgfkeysvalueof{`⟨*full key*⟩`}`

Inserts the value stored in ⟨*full key*⟩ at the current position into the text. It expands to an alias of the primitive `\relax` if the key is undefined.

Hello, world!

```
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysvalueof{/my family/my key}
```

Note: It is an error to assign to the result of the expansion of `\pgfkeysvalueof`, not only semantically but in recent versions of PGF also logically. To set the value of a key *always* use the appropriate interfaces, e.g. `\pgfkeyssetvalue`.

`\pgfkeysunexpandedvalueof{`⟨*full key*⟩`}`

This is the same as `\pgfkeysvalueof`, but the result will be wrapped inside `\unexpanded`.

\pgfkeysdetokenizedvalueof{⟨*full key*⟩}

This is the same as \pgfkeysvalueof, but the result will be wrapped inside \detokenize.

\pgfkeysifdefined{⟨*full key*⟩}{⟨*if*⟩}{⟨*else*⟩}

Checks whether this key was previously set using either \pgfkeyssetvalue or \pgfkeyslet. If so, the code in ⟨*if*⟩ is executed, otherwise the code in ⟨*else*⟩.

yes
```
\pgfkeyssetvalue{/my family/my key}{Hello, world!}
\pgfkeysifdefined{/my family/my key}{yes}{no}
```

\pgfkeysifassignable{⟨*full key*⟩}{⟨*if*⟩}{⟨*else*⟩}

Checks whether this key was previously set using either \pgfkeyssetvalue, \pgfkeyslet, \pgfkeysdef, or one of \pgfkeysdef's relatives. If so, the code in ⟨*if*⟩ is executed, otherwise the code in ⟨*else*⟩.

yesyes
```
\pgfkeysdef{/my family/my keyA}{\def\mykeyA{#1}}
\pgfkeyssetvalue{/my family/my keyB}{Hello, world!}
\pgfkeysifassignable{/my family/my keyA}{yes}{no}
\pgfkeysifassignable{/my family/my keyB}{yes}{no}
```

\pgfkeysifempty{⟨*full key*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

Tests whether the value stored in ⟨*full key*⟩ is empty. If it is empty, ⟨*true code*⟩ is executed, otherwise ⟨*false code*⟩ is executed.

The command is intended for keys that store values, as created for instance by the handler /.initial or by \pgfkeyssetvalue.

There is no label.The label is 'Hello'.

```
\pgfkeys{/my family/label/.initial=}

\pgfkeysifempty{/my family/label}
  {There is no label.}
  {The label is `\pgfkeysvalueof{/my family/label}'.}

\pgfkeys{/my family/label=Hello}

\pgfkeysifempty{/my family/label}
  {There is no label.}
  {The label is `\pgfkeysvalueof{/my family/label}'.}
```

Note: If the key passed to this macro does not exist, the behavior is *undefined*. Currently this macro uses \pgfkeysdetokenizedvalueof internally which expands to an alias of the \relax primitive when the key does not exist—therefore *a non-existent key is considered non-empty*. However, these semantics could change in the future and must not be relied upon. To be safe, check \pgfkeysifdefined first.

87.3 Setting Keys

Setting keys is done using a powerful command called \pgfkeys. This command takes a list of so-called *key–value pairs*. These are pairs of the form ⟨*key*⟩=⟨*value*⟩. The principal idea is the following: For each pair in the list, some *action* is taken. This action can be one of the following:

1. A command is executed whose argument(s) are ⟨*value*⟩. This command is stored in a special subkey of ⟨*key*⟩.
2. The ⟨*value*⟩ is stored in the ⟨*key*⟩ itself.
3. If the key's name (the part after the last slash) is a known *handler*, then this handler will take care of the key.
4. If the key is totally unknown, one of several possible *unknown key handlers* is called.

Additionally, if the ⟨*value*⟩ is missing, a default value may or may not be substituted. Before we plunge into all the details, let us have a quick look at the command itself.

\pgfkeys{⟨*key list*⟩}

The ⟨*key list*⟩ should be a list of key–value pairs, separated by commas. A key–value pair can have the following two forms: ⟨*key*⟩=⟨*value*⟩ or just ⟨*key*⟩. Any spaces around the ⟨*key*⟩ or around the ⟨*value*⟩ are removed. It is permissible to surround both the ⟨*key*⟩ or the ⟨*value*⟩ in curly braces, which are also removed. Especially putting the ⟨*value*⟩ in curly braces needs to be done quite often, namely whenever the ⟨*value*⟩ contains an equal-sign or a comma.

The key–value pairs in the list are handled in the order they appear. How this handling is done, exactly, is described in the rest of this section.

If a ⟨*key*⟩ is a partial key, the current value of the default path is prefixed to the ⟨*key*⟩ and this "upgraded" key is then used. The default path is just the root path `/` when the first key is handled, but it may change later on. At the end of the command, the default path is reset to the value it had before this command was executed.

Calls of this command may be nested. Thus, it is permissible to call `\pgfkeys` inside the code that is executed for a key. Since the default path is restored after a call of `\pgfkeys`, the default path will not change when you call `\pgfkeys` while executing code for a key (which is exactly what you want).

\pgfqkeys{⟨*default path*⟩}{⟨*key list*⟩}

This command has the same effect as `\pgfkeys{`⟨*default path*⟩`/.cd,`⟨*key list*⟩`}`, it is only marginally quicker. This command should not be used in user code, but rather in commands like `\tikzset` or `\pgfset` that get called very often.

\pgfkeysalso{⟨*key list*⟩}

This command has exactly the same effect as `\pgfkeys`, only the default path is not modified before or after the keys are being set. This command is mainly intended to be called by the code that is being processed for a key.

\pgfqkeysalso{⟨*default path*⟩}{⟨*key list*⟩}

This command has the same effect as `\pgfkeysalso{`⟨*default path*⟩`/.cd,`⟨*key list*⟩`}`, it is only quicker. Changing the default path inside a `\pgfkeyalso` is dangerous, so use with care. A rather safe place to call this command is at the beginning of a TEX group.

87.3.1 First Char Syntax Detection

Usually, keys are of the form ⟨*key*⟩=⟨*value*⟩ and how such keys are handled is discussed in the rest of this section. However, it is also possible to setup a different syntax for certain parts of the input to `\pgfkeys`. Since this is a rather advanced option, most readers may wish to skip the following discussion upon first reading; it is discussed here because this special syntax detection is the very first thing that is done when a key is processed, before any of the following operations are performed.

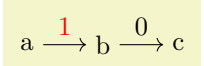
The `\pgfkeys` command and its variants decompose their input into a list of ⟨*string*⟩s that are separated by commas. By default, each such ⟨*string*⟩ must either have the form ⟨*key*⟩=⟨*value*⟩ or of the form ⟨*key*⟩ with the value-part missing. However, you might wish to interpret some of these strings differently. For instance, when a ⟨*string*⟩ has the form `"`⟨*text*⟩`"`, you might wish the ⟨*string*⟩ to be interpreted as if one had written `label text={`⟨*text*⟩`}`. Then, people could write

```
\myset{red, "main valve", thick}
```

instead of the more cumbersome

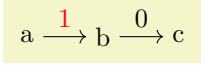
```
\myset{red, label text=main valve, thick}
```

An example where such a syntax reinterpretation is done is the `quotes` library, which allows you to write things like




```
\usetikzlibrary {graphs,quotes}
\tikz \graph { a ->["1" red] b ->["0"] c };
```

instead of the somewhat longer




```
\usetikzlibrary {graphs}
\tikz \graph { a ->[edge node={node[red,auto]{1}}] b ->[edge label=0] c };
```

In order to detect whether a ⟨*string*⟩ has a special syntax, you can request that the *first character* of ⟨*string*⟩ is analysed by the key parser. If this first character matches a character that has been flagged as a special character, the ⟨*string*⟩ is not interpreted as a usual key–value pair. Instead, ⟨*string*⟩ is passed as a parameter to a special macro that should take care of the ⟨*string*⟩. After this macro has finished, the parsing continues with the ⟨*next string*⟩ in the list.

In order to setup a special syntax handling for ⟨*strings*⟩ that begin with a certain character, two things need to be done:

1. First, the whole `first char syntax` detection must be "switched on", since, by default, it is turned off for efficiency reasons (the overhead is rather small, however). This is done by setting the following key:

   `/handlers/first char syntax=`⟨*true or false*⟩ (default `true`, initially `false`)

2. Second, in order to handle strings starting with a certain ⟨*character*⟩ in a special way, you need to store a macro in the following key:

   `/handlers/first char syntax/`⟨*meaning of character*⟩ (no value)

   The ⟨*meaning of character*⟩ should be the text that TEX's command `\meaning` returns for a macro that has been `\let` to the ⟨*character*⟩. For instance, when strings starting with `"` should be treated in a special way, the ⟨*meaning of character*⟩ would be the string `the character "` since this is what TEX writes when you say

   the character "
   ```
   \let\mycharacter="
   \meaning\mycharacter
   ```

   Now, the key `/handlers/first char syntax/`⟨*meaning of character*⟩ should be setup (using `\pgfkeyssetvalue` or using the `.initial` handler) to store a ⟨*macro name*⟩.

   If this is the case and if ⟨*string*⟩ starts with the ⟨*character*⟩ (blanks at the beginning of ⟨*string*⟩ are deleted prior to this test), then ⟨*macro name*⟩ is called with ⟨*string*⟩ as its argument.

Let us now have a look at an example. We install two handlers, one for strings starting with `"` and one for strings starting with `<`.

```
Quoted: "foo". Pointed: <bar>.
```

```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character "/.initial=\myquotemacro,
  /handlers/first char syntax/the character </.initial=\mypointedmacro,
}

\def\myquotemacro#1{Quoted: #1. }
\def\mypointedmacro#1{Pointed: #1. }

\ttfamily \pgfkeys{"foo", <bar>}
```

Naturally, in the above examples, the two handling macros did not do something particularly exciting. In the next example, we setup a more elaborate macro that mimics a small part the behavior of the `quotes` library, only for single quotes:

'foo'
bar

```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character '/.initial=\mysinglequotemacro
}

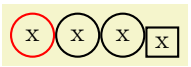
\def\mysinglequotemacro#1{\pgfkeysalso{label={#1}}}

\tikz \node [circle, 'foo', draw] {bar};
```

Note that in the above example, the macro `\mysinglequotemacro` gets passed the complete string, including the single quotes. It is the job of the macro to get rid of them, if this is necessary.

The first char syntax detection allows you to perform rather powerful transformations on the syntax of keys – provided you can "pin down" the syntax on the first character. In the following example, you can

write expressions in parentheses in front of a key–value pair and the pair will only be executed when the expression evaluates to true:




```
\pgfkeys{
  /handlers/first char syntax=true,
  /handlers/first char syntax/the character (/.initial=\myparamacro
}

\def\myparamacro#1{\myparaparser#1\someendtext}
\def\myparaparser(#1)#2\someendtext{
  \pgfmathparse{#1}
  \ifx\pgfmathresult\onetext
    \pgfkeysalso{#2}
  \fi
}
\def\onetext{1}

\foreach \i in {1,...,4}
  \tikz \node [draw, thick, rectangle, (pi>\i) circle, (pi>\i*2) draw=red] {x};
```

87.3.2 Default Arguments

The arguments of the `\pgfkeys` command can either be of the form ⟨*key*⟩=⟨*value*⟩ or of the form ⟨*key*⟩ with the value-part missing. In the second case, the `\pgfkeys` will try to provide a *default value* for the ⟨*value*⟩. If such a default value is defined, it will be used as if you had written ⟨*key*⟩=⟨*default value*⟩.

In the following, the details of how default values are determined is described; however, you should normally use the handlers `/.default` and `/.value required` as described in Section 87.4.2 and you may wish to skip the following details.

When `\pgfkeys` encounters a ⟨*key*⟩ without an equal-sign, the following happens:

1. The input is replaced by ⟨*key*⟩=`\pgfkeysnovalue`. In particular, the commands `\pgfkeys{my key}` and `\pgfkeys{my key=\pgfkeysnovalue}` have exactly the same effect and you can "simulate" a missing value by providing the value `\pgfkeysnovalue`, which is sometimes useful.
2. If the ⟨*value*⟩ is `\pgfkeysnovalue`, then it is checked whether the subkey ⟨*key*⟩`/.@def` exists. For instance, if you write `\pgfkeys{/my key}`, then it is checked whether the key `/my key/.@def` exists.
3. If the key ⟨*key*⟩`/.@def` exists, then the tokens stored in this key are used as ⟨*value*⟩.
4. If the key does not exist, then `\pgfkeysnovalue` is used as the ⟨*value*⟩.
5. At the end, if the ⟨*value*⟩ is now equal to `\pgfkeysvaluerequired`, then the code (or something fairly equivalent) `\pgfkeys{/errors/value required=`⟨*key*⟩`{}}` is executed. Thus, by changing this key you can change the error message that is printed or you can handle the missing value in some other way.

87.3.3 Keys That Execute Commands

After the transformation process described in the previous subsection, we arrive at a key of the form ⟨*key*⟩=⟨*value*⟩, where ⟨*key*⟩ is a full key. Different things can now happen, but always the macro `\pgfkeyscurrentkey` will have been set up to expand to the text of the ⟨*key*⟩ that is currently being processed.

The first things that is tested is whether the key ⟨*key*⟩`/.@cmd` exists. If this is the case, then it is assumed that this key stores the code of a macro and this macro is executed. The argument of this macro is ⟨*value*⟩ directly followed by `\pgfeov`, which stands for "end of value". The ⟨*value*⟩ is not surrounded by braces. After this code has been executed, `\pgfkeys` continues with the next key in the ⟨*key list*⟩.

It may seem quite peculiar that the macro stored in the key ⟨*key*⟩`/.@cmd` is not simply executed with the argument `{`⟨*value*⟩`}`. However, the approach taken in the `pgfkeys` packages allows for more flexibility. For instance, assume that you have a key that expects a ⟨*value*⟩ of the form "⟨*text*⟩`+`⟨*more text*⟩" and wishes to store ⟨*text*⟩ and ⟨*more text*⟩ in two different macros. This can be achieved as follows:

\a is 'hello', \b is 'world'.

```
\usepackage {shortvrb} \MakeShortVerb {\|}
\def\mystore#1+#2\pgfeov{\def\a{#1}\def\b{#2}}
\pgfkeyslet{/my key/.@cmd}{\mystore}
\pgfkeys{/my key=hello+world}

|\a| is `\a', |\b| is `\b'.
```

Naturally, defining the code to be stored in a key in the above manner is too awkward. The following commands simplify things a bit, but the usual manner of setting up code for a key is to use one of the handlers described in Section 87.4.3.

`\pgfkeysdef{`⟨*key*⟩`}{`⟨*code*⟩`}`

This command temporarily defines a TEX-macro with the argument list `#1\pgfeov` and then lets ⟨*key*⟩`/.@cmd` be equal to this macro. The net effect of all this is that you have then set up code for the key ⟨*key*⟩ so that when you write `\pgfkeys{`⟨*key*⟩`=`⟨*value*⟩`}`, then the ⟨*code*⟩ is executed with all occurrences of `#1` in ⟨*code*⟩ being replaced by ⟨*value*⟩. (This behavior is quite similar to the `\define@key` command of `keyval` and `xkeyval`).

hello, hello.

```
\pgfkeysdef{/my key}{#1, #1.}
\pgfkeys{/my key=hello}
```

`\pgfkeysedef{`⟨*key*⟩`}{`⟨*code*⟩`}`

This command works like `\pgfkeysdef`, but it uses `\edef` rather than `\def` when defining the key macro. If you do not know the difference between the two, then you will not need this command; and if you know the difference, then you will know when you need this command.

`\pgfkeysdefnargs{`⟨*key*⟩`}{`⟨*argument count*⟩`}{`⟨*code*⟩`}`

This command works like `\pgfkeysdef`, but it allows you to provide an arbitrary ⟨*argument count*⟩ between 0 and 9 (inclusive).

`\a` is 'hello', `\b` is 'world'.

```
\usepackage {shortvrb} \MakeShortVerb {\|}
\pgfkeysdefnargs{/my key}{2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key=
    {hello}
    {world}}

|\a| is `\a', |\b| is `\b'.
```

The resulting key will expect exactly `{`⟨*argument count*⟩`}` arguments.

`\pgfkeysedefnargs{`⟨*key*⟩`}{`⟨*argument count*⟩`}{`⟨*code*⟩`}`

The `\edef` version of `\pgfkeysdefnargs`.

`\pgfkeysdefargs{`⟨*key*⟩`}{`⟨*argument pattern*⟩`}{`⟨*code*⟩`}`

This command works like `\pgfkeysdefnargs`, but it allows you to provide an arbitrary ⟨*argument pattern*⟩ rather than just a number of arguments.

`\a` is 'hello', `\b` is 'world'.

```
\usepackage {shortvrb} \MakeShortVerb {\|}
\pgfkeysdefargs{/my key}{#1+#2}{\def\a{#1}\def\b{#2}}
\pgfkeys{/my key=hello+world}

|\a| is `\a', |\b| is `\b'.
```

Note that `\pgfkeysdefnargs` is *better* when it comes to simple argument *counts*[14].

`\pgfkeysedefargs{`⟨*key*⟩`}{`⟨*argument pattern*⟩`}{`⟨*code*⟩`}`

The `\edef` version of `\pgfkeysdefargs`.

87.3.4 Keys That Store Values

Let us continue with what happens when `\pgfkeys` processes the current key and the subkey ⟨*key*⟩`/.@cmd` is not defined. Then it is checked whether the ⟨*key*⟩ itself exists (has been previously assigned a value using, for instance, `\pgfkeyssetvalue`). In this case, the tokens stored in ⟨*key*⟩ are replaced by ⟨*value*⟩ and `\pgfkeys` proceeds with the next key in the ⟨*key list*⟩.

[14] When the resulting keys are used, the `defnargs` variant allows spaces between arguments whereas the `defargs` variant does not; it considers the spaces as part of the argument.

87.3.5 Keys That Are Handled

If neither the ⟨*key*⟩ itself nor the subkey ⟨*key*⟩`/.@cmd` are defined, then the ⟨*key*⟩ cannot be processed "all by itself". Rather, a ⟨*handler*⟩ is needed for this key. Most of the power of `pgfkeys` comes from the proper use of such handlers.

Recall that the ⟨*key*⟩ is always a full key (if it was not originally, it has already been upgraded at this point to a full key). It decomposed into two parts:

1. The ⟨*path*⟩ of ⟨*key*⟩ (everything before the last slash) is stored in the macro `\pgfkeyscurrentpath`.
2. The ⟨*name*⟩ of ⟨*key*⟩ (everything after the last slash) is stored in the macro `\pgfkeyscurrentname`.

   It is recommended (but not necessary) that the name of a handler starts with a dot (but not with `.@`), so that they are easy to detect for the reader.

(For efficiency reasons, these two macros are only set up at this point; so when code is executed for a key in the "usual" manner then these macros are not set up.)

The `\pgfkeys` command now checks whether the key `/handlers/`⟨*name*⟩`/.@cmd` exists. If so, it should store a command and this command is executed exactly in the same manner as described in Section 87.3.3. Thus, this code gets the ⟨*value*⟩ that was originally intended for ⟨*key*⟩ as its argument, followed by `\pgfeov`. It is the job of the handlers to do something useful with the ⟨*value*⟩.

For an example, let us write a handler that will output the value stored in a key to the log file. We call this handler `/.print to log`. The idea is that when someone tries to use the key `/my key/.print to log`, then this key will not be defined and the handler gets executed. The handler will then have access to the path-part of the key, which is `/my key`, via the macro `\pgfkeyscurrentpath`. It can then lookup which value is stored in this key and print it.

```
\pgfkeysdef{/handlers/.print to log}
{%
  \pgfkeysgetvalue{\pgfkeyscurrentpath}{\temp}
  \writetolog{\temp}
}
\pgfkeyssetvalue{/my key}{Hi!}
...
\pgfkeys{/my key/.print to log}
```

The above code will print `Hi!` in the log, provided the macro `\writetolog` is set up appropriately.

For a more interesting handler, let us program a handler that will set up a key so that when the key is used, some code is executed. This code is given as ⟨*value*⟩. All the handler must do is to call `\pgfkeysdef` for the path of the key (which misses the handler's name) and assign the parameter value to it.

(hallo)

```
\pgfkeysdef{/handlers/.my code}{\pgfkeysdef{\pgfkeyscurrentpath}{#1}}
\pgfkeys{/my key/.my code=(#1)}
\pgfkeys{/my key=hallo}
```

There are some parameters for handled keys which prove to be useful in some (possibly rare) special cases:

`/handler config=all|only existing|full or existing` (no default, initially `all`)

Changes the initial configuration how key handlers will be used.

This configuration is for advanced users and rarely necessary.

`all` The preconfigured setting `all` works as described above and imposes no restriction on the key setting process.

`only existing` The value `only existing` modifies the algorithm for handled keys as follows: a handler ⟨*key name*⟩`/.`⟨*handler*⟩ will be executed only if ⟨*key name*⟩ is either a key which stores its value directly or a command key for which `/.@cmd` exists. If ⟨*key name*⟩ does *not* exist already, the complete string ⟨*key name*⟩`/.`⟨*handler*⟩ is considered to be an unknown key and the procedure described in the next section applies (for the path of ⟨*key name*⟩).

Initial definition.Re-Definition.Unknown key '/the/other key/.code'.

```
% Define a test key and error handlers:
\pgfkeys{/the/key/.code={Initial definition. }}
\pgfkeys{/handlers/.unknown/.code={Unknown key `\pgfkeyscurrentkey'. }}

% calling the test key yields 'Initial definition. ':
\pgfkeys{/the/key}

% Change configuration:
\pgfkeys{/handler config=only existing}

% allowed: key *re*-definition:
\pgfkeys{/the/key/.code={Re-Definition. }}
% calling the key yields 'Re-Definition. ':
\pgfkeys{/the/key}

% not allowed: definition of new keys:
% this checks for '/the/other key/.unknown'
% and '/handlers/.unknown'
% and yields finally
% 'Unknown key `/the/other key/.code`'
\pgfkeys{/the/other key/.code={New definition. }}
```

It is necessary to exclude some key handlers from this procedure. Altogether, the detailed procedure is as follows:

1. If a handled key like `/a path/a key/.a handler=value` is encountered, it is checked whether the handler should be invoked. This is the case if
   - An exception from `only existing` for this key exists (see below),
   - The key `/a path/a key` exists already – either directly as storage key or with the `.@cmd` suffix.
2. If the check passes, everything works as before.
3. If the check fails, the complete key will be considered to be unknown. In that case, the handling of unknown keys as described in the next section applies. There, the current key path will be set to `/a path` and the current key's name to `key/.a handler`.

A consequence of this configuration is to provide more meaningful processing of handled keys if a search path for keys is in effect, see section 87.3.6 for an example.

`full or existing` Finally, the choice `full or existing` is a variant of `only existing`: it works in the same way for keys which do not have a full key path. For example, the style

`\pgfkeys{/my path/.cd,key/.style={...}}`

can only be redefined: it doesn't have a full path, so the `only existing` mechanism applies. But the style

`\pgfkeys{/my path/key/.style={...}}`

will still work. This allows users to override the `only existing` feature if they know what they're doing (and provide full key paths).

`/handler config/only existing/add exception={`⟨*key handler name*⟩`}` (no default)

Allows to add exceptions to the `/handler config=only existing` feature. Initially exceptions for the key handlers `/.cd`, `/.try`, `/.retry`, `/.lastretry` and `/.unknown` are defined. The value `{`⟨*key handler name*⟩`}` should be the name of a key handler.

87.3.6 Keys That Are Unknown

For some keys, neither the key, nor its `.@cmd` subkey nor a handler is defined. In this case, it is checked whether the key ⟨*current path*⟩`/.unknown/.@cmd` exists. Thus, when you try to use the key `/tikz/strange`, then it is checked whether `/tikz/.unknown/.@cmd` exists. If this key exists (which it does), it is executed. This code can then try to make sense of the key. For instance, the handler for Ti*k*Z will try to interpret the key's name as a color or as an arrow specification or as a PGF option.

You can set up unknown key handlers for your own keys by simply setting the code of the key ⟨*my path prefix*⟩`/.unknown`. This also allows you to set up "search paths". The idea is that you would like keys to be searched not only in a single default path, but in several. Suppose, for instance, that you would like keys to be searched for in `/a`, `/b`, and `/b/c`. We set up a key `/my search path` for this:

```
\pgfkeys{/my search path/.unknown/.code=
  {%
    \let\searchname=\pgfkeyscurrentname%
    \pgfkeysalso{%
      /a/\searchname/.try=#1,
      /b/\searchname/.retry=#1,
      /b/c/\searchname/.retry=#1%
    }%
  }%
}
\pgfkeys{/my search path/.cd,foo,bar}
```

In the above code, `foo` and `bar` will be searched for in the three directories `/a`, `/b`, and `/b/c`. Before you start implementing search paths using this pattern, consider the `/.search also` handler discussed below.

If the key ⟨*current path*⟩`/.unknown/.@cmd` does not exist, the handler `/handlers/.unknown` is invoked instead, which is always defined and which prints an error message by default.

87.3.7 Search Paths And Handled Keys

There is one special case which occurs in the search path example above. What happens if we want to change a style? For example,

```
\pgfkeys{/my search path/.cd,custom/.style={variables}}
```

could mean a style in `/my search path/`, `/a/`, `/b/` or even `/b/c/`!

Due to the rules for handled keys, the answer is `/my search path/custom/.style={variables}`. It may be useful to modify this default behavior. One useful thing would be to search for *existing* styles named `custom` and redefine them. For example, if a style `/b/custom` exists, the assignment `custom/.style={variables}` should probably redefine `/b/custom` instead of `/my search path/custom`. This can be done using `handler config`:

This is ‘/b/custom’. This is ‘/b/custom’.Modified.

```
\pgfkeys{/my search path/.unknown/.code=
  {%
    \let\searchname=\pgfkeyscurrentname%
    \pgfkeysalso{%
      /a/\searchname/.try=#1,
      /b/\searchname/.retry=#1,
      /b/c/\searchname/.retry=#1%
    }%
  }%
}

% Let's define /b/custom here:
\pgfkeys{/b/custom/.code={This is `\pgfkeyscurrentkey'. }}

% Reconfigure treatment of key handlers:
\pgfkeys{/handler config=only existing}

% The search path procedure will find /b/custom
% -> leads to This is `/b/custom'
\pgfkeys{/my search path/.cd,custom}

% Due to the reconfiguration, this will find /b/custom instead of
% defining /my search path/custom:
\pgfkeys{/my search path/.cd,custom/.append code={Modified. }}

% So using the search path, we again find /b/custom which
% leads to This is `/b/custom' Modified
\pgfkeys{/my search path/.cd,custom}
```

A slightly different approach to search paths can be realized using the `/.search also` key handler, see below.

87.4 Key Handlers

We now describe which key handlers are defined by default. You can also define new ones as described in Section 87.3.5.

87.4.1 Handlers for Path Management

Key handler ⟨*key*⟩`/.cd`

This handler causes the default path to be set to ⟨*key*⟩. Note that the default path is reset at the beginning of each call to `\pgfkeys` to be equal to `/`.

Example: `\pgfkeys{/tikz/.cd,...}`

Key handler ⟨*key*⟩`/.is family`

This handler sets up things such that when ⟨*key*⟩ is executed, then the current path is set to ⟨*key*⟩. A typical use is the following:

```
\pgfkeys{/tikz/.is family}
\pgfkeys{tikz,line width=1cm}
```

The effect of this handler is the same as if you had written ⟨*key*⟩`/.style=`⟨*key*⟩`/.cd`, only the code produced by the `/.is family` handler is quicker.

87.4.2 Setting Defaults

Key handler ⟨*key*⟩`/.default=`⟨*value*⟩

Sets the default value of ⟨*key*⟩ to ⟨*value*⟩. This means that whenever no value is provided in a call to `\pgfkeys`, then this ⟨*value*⟩ will be used instead.

Example: `\pgfkeys{/width/.default=1cm}`

Key handler ⟨*key*⟩`/.value required`

This handler causes the error message key `/errors/value required` to be issued whenever the ⟨*key*⟩ is used without a value.

Example: `\pgfkeys{/width/.value required}`

Key handler ⟨*key*⟩`/.value forbidden`

This handler causes the error message key `/errors/value forbidden` to be issued whenever the ⟨*key*⟩ is used with a value.

This handler works be adding code to the code of the key. This means that you have to define the key first before you can use this handler.

```
\pgfkeys{/my key/.code=I do not want an argument!}
\pgfkeys{/my key/.value forbidden}

\pgfkeys{/my key}     % Ok
\pgfkeys{/my key=foo} % Error
```

87.4.3 Defining Key Codes

A number of handlers exist for defining the code of keys.

Key handler ⟨*key*⟩`/.code=`⟨*code*⟩

This handler executes `\pgfkeysdef` with the parameters ⟨*key*⟩ and ⟨*code*⟩. This means that, afterwards, whenever the ⟨*key*⟩ is used, the ⟨*code*⟩ gets executed. More precisely, when ⟨*key*⟩`=`⟨*value*⟩ is encountered in a key list, ⟨*code*⟩ is executed with any occurrence of `#1` replaced by ⟨*value*⟩. As always, if no ⟨*value*⟩ is given, the default value is used, if defined, or the special value `\pgfkeysnovalue`.

It is permissible that ⟨*code*⟩ calls the command `\pgfkeys`. It is also permissible that ⟨*code*⟩ calls the command `\pgfkeysalso`, which is useful for styles, see below.

```
\pgfkeys{/par indent/.code={\parindent=#1},/par indent/.default=2em}
\pgfkeys{/par indent=1cm}
...
\pgfkeys{/par indent}
```

Key handler ⟨*key*⟩`/.ecode=`⟨*code*⟩

This handler works like `/.code`, only the command `\pgfkeysedef` is used.

Key handler ⟨*key*⟩`/.code 2 args=`⟨*code*⟩

This handler works like `/.code`, only two arguments rather than one are expected when the ⟨*code*⟩ is executed. This means that when ⟨*key*⟩=⟨*value*⟩ is encountered in a key list, the ⟨*value*⟩ should consist of two arguments. For instance, ⟨*value*⟩ could be `{first}{second}`. Then ⟨*code*⟩ is executed with any occurrence of `#1` replaced `first` and any occurrence of `#2` replaced by `second`.

```
\pgfkeys{/page size/.code 2 args={\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size={30cm}{20cm}}
```

Because of the special way the ⟨*value*⟩ is parsed, if you set ⟨*value*⟩ to, for instance, `first` (without any braces), then `#1` will be set to `f` and `#2` will be set to `irst`.

Key handler ⟨*key*⟩`/.ecode 2 args=`⟨*code*⟩

This handler works like `/.code 2 args`, only an `\edef` is used rather than a `\def` to define the macro.

Key handler ⟨*key*⟩`/.code n args={`⟨*argument count*⟩`}{`⟨*code*⟩`}`

This handler also works like `/.code`, but you can now specify a number of arguments between 0 and 9 (inclusive).

First=‘A’, Second=‘B’

```
\pgfkeys{/a key/.code n args={2}{First=`#1', Second=`#2'}}
\pgfkeys{/a key={A}{B}}
```

In contrast to `/.code 2 args`, there must be exactly ⟨*argument count*⟩ arguments, not more and not less and these arguments should be properly delimited.

Key handler ⟨*key*⟩`/.ecode n args={`⟨*argument count*⟩`}{`⟨*code*⟩`}`

This handler works like `/.code n args`, only an `\edef` is used rather than a `\def` to define the macro.

Key handler ⟨*key*⟩`/.code args={`⟨*argument pattern*⟩`}{`⟨*code*⟩`}`

This handler is the most flexible way to define a `/.code` key: you can now specify an arbitrary ⟨*argument pattern*⟩. Such a pattern is a usual TEX macro pattern. For instance, suppose ⟨*argument pattern*⟩ is `(#1/#2)` and ⟨*key*⟩=⟨*value*⟩ is encountered in a key list with ⟨*value*⟩ being `(first/second)`. Then ⟨*code*⟩ is executed with any occurrence of `#1` replaced `first` and any occurrence of `#2` replaced by `second`. So, the actual ⟨*value*⟩ is matched against the ⟨*argument pattern*⟩ in the standard TEX way.

```
\pgfkeys{/page size/.code args={#1 and #2}{\paperheight=#2\paperwidth=#1}}
\pgfkeys{/page size=30cm and 20cm}
```

Note that `/.code n args` should be preferred in case you need just a number of arguments (when the resulting keys are used, `/.code n args` gobbles spaces between the arguments whereas `/.code args` considers spaces to be part of the argument).

Key handler ⟨*key*⟩`/.ecode args={`⟨*argument pattern*⟩`}{`⟨*code*⟩`}`

This handler works like `/.code args`, only an `\edef` is used rather than a `\def` to define the macro.

There are also handlers for modifying existing keys.

Key handler ⟨*key*⟩`/.add code={`⟨*prefix code*⟩`}{`⟨*append code*⟩`}`

This handler adds code to an existing key. The ⟨*prefix code*⟩ is added to the code stored in ⟨*key*⟩`/.@cmd` at the beginning, the ⟨*append code*⟩ is added to this code at the end. Either can be empty. The argument list of ⟨*code*⟩ cannot be changed using this handler. Note that both ⟨*prefix code*⟩ and ⟨*append code*⟩ may contain parameters like `#2`.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\newdimen\myparindent
\pgfkeys{/par indent/.add code={}{\myparindent=#1}}
...
\pgfkeys{/par indent=1cm} % This will set both \parindent and
                          % \myparindent to 1cm
```

Key handler ⟨*key*⟩`/.prefix code=`⟨*prefix code*⟩

This handler is a shortcut for ⟨*key*⟩`/.add code={`⟨*prefix code*⟩`}{}`. That is, this handler adds the ⟨*prefix code*⟩ at the beginning of the code stored in ⟨*key*⟩`/.@cmd`.

Key handler ⟨*key*⟩`/.append code=`⟨*append code*⟩

This handler is a shortcut for ⟨*key*⟩`/.add code={}{`⟨*append code*⟩`}`.

87.4.4 Defining Styles

The following handlers allow you to define *styles*. A style is a key list that is processed whenever the style is given as a key in a key list. Thus, a style "stands for" a certain key value list. Styles can be parameterized just like normal code.

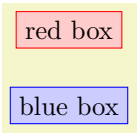
Key handler ⟨*key*⟩`/.style=`⟨*key list*⟩

This handler sets things up so that whenever ⟨*key*⟩=⟨*value*⟩ is encountered in a key list, then the ⟨*key list*⟩, with every occurrence of `#1` replaced by ⟨*value*⟩, is processed instead. As always, if no ⟨*value*⟩ is given, the default value is used, if defined, or the special value `\pgfkeysnovalue`.

You can achieve the same effect by writing ⟨*key*⟩`/.code=\pgfkeysalso{`⟨*key list*⟩`}`. This means, in particular, that the code of a key could also first execute some normal code and only then process some further keys.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\pgfkeys{/no indent/.style={/par indent=0pt}}
\pgfkeys{/normal indent/.style={/par indent=2em}}
\pgfkeys{/no indent}
...
\pgfkeys{/normal indent}
```

The following example shows a parameterized style "in action".




```
\begin{tikzpicture}[outline/.style={draw=#1,fill=#1!20}]
  \node [outline=red]            {red box};
  \node [outline=blue] at (0,-1) {blue box};
\end{tikzpicture}
```

Key handler ⟨*key*⟩`/.estyle=`⟨*key list*⟩

This handler works like `/.style`, only the ⟨*code*⟩ is set using `\edef` rather than `\def`. Thus, all macros in the ⟨*code*⟩ are expanded prior to saving the style.

For styles the corresponding handlers as for normal code exist:

Key handler ⟨*key*⟩`/.style 2 args=`⟨*key list*⟩

This handler works like `/.code 2 args`, only for styles. Thus, the ⟨*key list*⟩ may contain occurrences of both `#1` and `#2` and when the style is used, two parameters must be given as ⟨*value*⟩.

```
\pgfkeys{/paper height/.code={\paperheight=#1},/paper width/.code={\paperwidth=#1}}
\pgfkeys{/page size/.style 2 args={/paper height=#1,/paper width=#2}}
\pgfkeys{/page size={30cm}{20cm}}
```

Key handler ⟨*key*⟩`/.estyle 2 args=`⟨*key list*⟩

This handler works like `/.style 2 args`, only an `\edef` is used rather than a `\def` to define the macro.

Key handler ⟨*key*⟩`/.style n args={`⟨*argument count*⟩`}`⟨*key list*⟩

This handler works like `/.code n args`, only for styles. Here, ⟨*key list*⟩ may depend on all ⟨*argument count*⟩ parameters.

Key handler ⟨*key*⟩`/.add style={`⟨*prefix key list*⟩`}{`⟨*append key list*⟩`}`

This handler works like `/.add code`, only for styles. However, it is permissible to add styles to keys that have previously been set using `/.code`. (It is also permissible to add normal ⟨*code*⟩ to a key that

has previously been set using `/.style`). When you add a style to a key that was previously set using `/.code`, the following happens: When ⟨*key*⟩ is processed, the ⟨*prefix key list*⟩ will be processed first, then the ⟨*code*⟩ that was previously stored in ⟨*key*⟩`/.@cmd`, and then the keys in ⟨*append key list*⟩ are processed.

```
\pgfkeys{/par indent/.code={\parindent=#1}}
\pgfkeys{/par indent/.add style={}{/my key=#1}}
...
\pgfkeys{/par indent=1cm} % This will set \parindent and
                          % then execute /my key=#1
```

Key handler ⟨*key*⟩`/.style args={`⟨*argument pattern*⟩`}{`⟨*key list*⟩`}`

This handler works like `/.code args`, only for styles.

Key handler ⟨*key*⟩`/.estyle args={`⟨*argument pattern*⟩`}{`⟨*code*⟩`}`

This handler works like `/.ecode args`, only for styles.

Key handler ⟨*key*⟩`/.prefix style=`⟨*prefix key list*⟩

Works like `/.add style`, but only for the prefix key list.

Key handler ⟨*key*⟩`/.append style=`⟨*append key list*⟩

Works like `/.add style`, but only for the append key list.

87.4.5 Defining Value-, Macro-, If- and Choice-Keys

For some keys, the code that should be executed for them is rather "specialized". For instance, it happens often that the code for a key just sets a certain TEX-if to true or false. For these cases, predefined handlers make it easier to install the necessary code.

However, we start with some handlers that are used to manage the value that is directly stored in a key.

Key handler ⟨*key*⟩`/.initial=`⟨*value*⟩

This handler sets the value of ⟨*key*⟩ to ⟨*value*⟩. Note that no subkeys are involved. After this handler has been used, by the rules governing keys, you can subsequently change the value of the ⟨*key*⟩ by just writing ⟨*key*⟩=⟨*value*⟩. Thus, this handler is used to set the initial value of key.

```
\pgfkeys{/my key/.initial=red}
% "/my key" now stores the value "red"
\pgfkeys{/my key=blue}
% "/my key" now stores the value "blue"
```

Note that with this configuration, writing `\pgfkeys{/my key}` will not have the effect you might expect (namely that `blue` is inserted into the main text). Rather, `/my key` will be promoted to `/my key=\pgfkeysnovalue` and, thus, `\pgfkeysnovalue` will be stored in `/my key`.

To retrieve the value stored in a key, the handler `/.get` is used.

Remark: A key can both store a value and execute commands[15]. If so, using ⟨*key*⟩ will *always* execute commands with the passed value, or the default value `\pgfkeysnovalue` if no value is provided. Note that the stored value is never used. To update the stored value, the handler `/.initial` or command `\pgfkeyssetvalue` is used.

blue

```
\pgfkeys{/my key/.initial=red}
\pgfkeys{/my key/.code=#1}
% "/my key" now both stores the value "red" and executes commands
\pgfkeys{/my key=blue}
% "/my key" now still stores the value "red"
```

Key handler ⟨*key*⟩`/.get=`⟨*macro*⟩

Executes a `\let` command so that ⟨*macro*⟩ contains the contents stored in ⟨*key*⟩.

[15] This behavior was partially changed in PGF 3.1.6 and then restored in 3.1.7. For compatibility reasons, this behavior will not be changed in future releases anymore.

blue

```
\pgfkeys{/my key/.initial=red}
\pgfkeys{/my key=blue}
\pgfkeys{/my key/.get=\mymacro}
\mymacro
```

Key handler ⟨*key*⟩`/.add={`⟨*prefix value*⟩`}{`⟨*append value*⟩`}`

Adds the ⟨*prefix value*⟩ at the beginning and the ⟨*append value*⟩ at the end of the value stored in ⟨*key*⟩.

Key handler ⟨*key*⟩`/.prefix={`⟨*prefix value*⟩`}`

Adds the ⟨*prefix value*⟩ at the beginning of the value stored in ⟨*key*⟩.

Key handler ⟨*key*⟩`/.append={`⟨*append value*⟩`}`

Adds the ⟨*append value*⟩ at the end of the value stored in ⟨*key*⟩.

Key handler ⟨*key*⟩`/.link=`⟨*another key*⟩

Stores the value `\pgfkeysvalueof{`⟨*another key*⟩`}` in the ⟨*key*⟩. The idea is that when you expand the ⟨*key*⟩, the value of ⟨*another key*⟩ is expanded instead. This corresponds loosely to the notion of soft links in Unix, hence the name.

The next handler is useful for the common situation where ⟨*key*⟩=⟨*value*⟩ should cause the ⟨*value*⟩ to be stored in some macro. Note that, typically, you could just as well store the value in the key itself.

Key handler ⟨*key*⟩`/.store in=`⟨*macro*⟩

This handler has the following effect: When you write ⟨*key*⟩=⟨*value*⟩, the code `\def`⟨*macro*⟩`{`⟨*value*⟩`}` is executed. Thus, the given value is "stored" in the ⟨*macro*⟩.

Hello Gruffalo!

```
\pgfkeys{/text/.store in=\mytext}
\def\a{world}
\pgfkeys{/text=Hello \a!}
\def\a{Gruffalo}
\mytext
```

Key handler ⟨*key*⟩`/.estore in=`⟨*macro*⟩

This handler is similar to `/.store in`, only the code `\edef`⟨*macro*⟩`{`⟨*value*⟩`}` is used. Thus, the macro-expanded version of ⟨*value*⟩ is stored in the ⟨*macro*⟩.

Hello world!

```
\pgfkeys{/text/.estore in=\mytext}
\def\a{world}
\pgfkeys{/text=Hello \a!}
\def\a{Gruffalo}
\mytext
```

In another common situation a key is used to set a TEX-if to true or false.

Key handler ⟨*key*⟩`/.is if=`⟨*TEX-if name*⟩

This handler has the following effect: When you write ⟨*key*⟩=⟨*value*⟩, it is first checked that ⟨*value*⟩ is `true` or `false` (the default is `true` if no ⟨*value*⟩ is given). If this is not the case, the error key `/errors/boolean expected` is executed. Otherwise, the code `\`⟨*TEX-if name*⟩⟨*value*⟩ is executed, which sets the TEX-if accordingly.

Round?

```
\newif\iftheworldisflat
\pgfkeys{/flat world/.is if=theworldisflat}
\pgfkeys{/flat world=false}
\iftheworldisflat
  Flat
\else
  Round?
\fi
```

The next handler deals with the problem when a ⟨*key*⟩=⟨*value*⟩ makes sense only for a small set of possible ⟨*value*⟩s. For instance, the line cap can only be `rounded` or `rect` or `butt`, but nothing else. For this situation the following handler is useful.

Key handler ⟨*key*⟩`/.is choice`

This handler sets things up so that writing ⟨*key*⟩=⟨*value*⟩ will cause the subkey ⟨*key*⟩/⟨*value*⟩ to be executed. So, each of the different possible choices should be given by a subkey of ⟨*key*⟩.

```
\pgfkeys{/line cap/.is choice}
\pgfkeys{/line cap/round/.code={\pgfsetroundcap}}
\pgfkeys{/line cap/butt/.code={\pgfsetbuttcap}}
\pgfkeys{/line cap/rect/.code={\pgfsetrectcap}}
\pgfkeys{/line cap/rectangle/.style={/line cap=rect}}
...
\draw [/line cap=butt] ...
```

If the subkey ⟨*key*⟩/⟨*value*⟩ does not exist, the error key `/errors/unknown choice value` is executed.

87.4.6 Expanded and Multiple Values

When you write ⟨*key*⟩=⟨*value*⟩, you usually wish to use the ⟨*value*⟩ "as is". Indeed, great care is taken to ensure that you can even use things like `#1` or unbalanced TEX-ifs inside ⟨*value*⟩. However, sometimes you want the ⟨*value*⟩ to be expanded before it is used. For instance, ⟨*value*⟩ might be a macro name like `\mymacro` and you do not want `\mymacro` to be used as the macro, but rather the *contents* of `\mymacro`. Thus, instead of using ⟨*value*⟩, you wish to use whatever ⟨*value*⟩ expands to. Instead of using some fancy `\expandafter` hackery, you can use the following handlers:

Key handler ⟨*key*⟩`/.expand once`=⟨*value*⟩

This handler expands ⟨*value*⟩ once (more precisely, it executes an `\expandafter` command on the first token of ⟨*value*⟩) and then process the resulting ⟨*result*⟩ as if you had written ⟨*key*⟩=⟨*result*⟩. Note that if ⟨*key*⟩ contains a handler itself, this handler will be called normally.

Key 1:	\c
Key 2:	\b
Key 3:	\a
Key 4:	bottom

```
\def\a{bottom}
\def\b{\a}
\def\c{\b}

\pgfkeys{/key1/.initial=\c}
\pgfkeys{/key2/.initial/.expand once=\c}
\pgfkeys{/key3/.initial/.expand twice=\c}
\pgfkeys{/key4/.initial/.expanded=\c}

\def\a{{\ttfamily\string\a}}
\def\b{{\ttfamily\string\b}}
\def\c{{\ttfamily\string\c}}

\begin{tabular}{ll}
Key 1:& \pgfkeys{/key1} \\
Key 2:& \pgfkeys{/key2} \\
Key 3:& \pgfkeys{/key3} \\
Key 4:& \pgfkeys{/key4}
\end{tabular}
```

Key handler ⟨*key*⟩`/.expand twice`=⟨*value*⟩

This handler works like saying ⟨*key*⟩`/.expand once/.expand once`=⟨*value*⟩.

Key handler ⟨*key*⟩`/.expanded`=⟨*value*⟩

This handler will completely expand ⟨*value*⟩ (using `\edef`) before processing ⟨*key*⟩=⟨*result*⟩.

Key handler ⟨*key*⟩`/.evaluated`=⟨*value*⟩

This handler will evaluate ⟨*value*⟩ as a mathematical expression with `\pgfmathparse` and assign ⟨*key*⟩=`\pgfmathresult`.

1.61803

```
\pgfkeys{
  /golden ratio/.initial/.evaluated={(1 + sqrt(5))/2},
}
\pgfkeys{/golden ratio}
```

Key handler ⟨*key*⟩`/.list=`⟨*comma-separated list of values*⟩

This handler causes the key to be used repeatedly, namely once for every element of the list of values. Note that the list of values should typically be surrounded by braces since, otherwise, TEX will not be able to tell whether a comma starts a new key or a new value.

The ⟨*list of values*⟩ is processed using the `\foreach` statement, so you can use the `...` notation.

(a)(b)(0)(1)(2)(3)(4)(5)

```
\pgfkeys{/foo/.code=(#1)}
\pgfkeys{/foo/.list={a,b,0,1,...,5}}
```

87.4.7 Handlers for Forwarding

Key handler ⟨*key*⟩`/.forward to=`⟨*another key*⟩

This handler causes the ⟨*key*⟩ to "forward" its argument to ⟨*another key*⟩. When the ⟨*key*⟩ is used, its normal code will be executed first. Then, the value is (additionally) passed to ⟨*another key*⟩. If the ⟨*key*⟩ has not yet been defined prior to the use of `.forward to`, it will be defined then (and do nothing by itself, expect for forwarding it to ⟨*key name*⟩). The ⟨*another key*⟩ must be a fully qualified key name.

(b:1)(a:1) (a:2)

```
\pgfkeys{
  /a/.code=(a:#1),
  /b/.code=(b:#1),
  /b/.forward to=/a,
  /c/.forward to=/a
}
\pgfkeys{/b=1} \pgfkeys{/c=2}
```

Key handler ⟨*key*⟩`/.search also={`⟨*path list*⟩`}`

A style which installs a `/.unknown` handler into ⟨*key*⟩. This `/.unknown` handler will then search for unknown keys in every path provided in `{`⟨*path list*⟩`}`.

Invoking /secondary path/option with 'value'

```
% define a key:
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with `#1'}}

% set up a search path:
\pgfkeys{/main path/.search also={/secondary path}}

% try searching for `option=value' in '/main path':
% -> this finds `/secondary path/option'!
\pgfkeys{/main path/.cd,option=value}
```

The `/.search also` handler follows the strategy

1. If a user provides a fully qualified key which could not be found, for example the full string `/main path/option`, it assumes that the user knew what she is doing – and does *not* continue searching for `option` in `{`⟨*path list*⟩`}`.
2. If a user provides only the key's name, for example `option` and `option` cannot be found in the current default path (which is `/main path` in our example above), the current default path is set to the next element in `{`⟨*path list*⟩`}` (which is `/secondary path` here) and `\pgfkeys` will be restarted. This will be iterated until either a match has been found or all elements in `{`⟨*path list*⟩`}` have been tested.
3. If all elements in `{`⟨*path list*⟩`}` have been checked and the key is still unknown, the fall-back handler `/handlers/.unknown` will be invoked.

Invoking /secondary path/option with 'value'Found unknown option /main path/option=value!

```
% define a key:
\pgfkeys{/secondary path/option/.code={Invoking /secondary path/option with `#1'}}

% set up a search path:
\pgfkeys{/main path/.search also={/secondary path}}

% try searching for `option=value' in '/main path':
% -> this finds `/secondary path/option'!
\pgfkeys{/main path/.cd,option=value}

% negative example:
% try searching for fully qualified key /main path/option.
% This won't be handled by .search also.
\pgfkeys{/handlers/.unknown/.code={Found unknown option \pgfkeyscurrentkeyRAW={#1}!}}%
\pgfkeys{/main path/.cd,/main path/option=value}
```

Please note that the strategy of `/.search also` is different from the first example provided in section 87.3.6 “Unknown Keys” because `/.search also` only applies to keys that are not fully qualified.

For those who are familiar with `\pgfkeys`, the actual implementation of `/.search also` might be interesting:

1. `\pgfkeys{/path/.search also={/tikz}}` is equivalent to

```
\pgfkeys{/path/.unknown/.code={%
        \def\pgfkeys@searchalso@temp@value{#1}%
        \ifpgfkeysaddeddefaultpath
            \expandafter\pgfkeys@firstoftwo
        \else
            \expandafter\pgfkeys@secondoftwo
        \fi{%
            % only process keys for which no full path has been
            % provided:
            \pgfkeyssuccessfalse
            \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
            \ifpgfkeyssuccess
            \else
                % search with /tikz as default path:
                \pgfqkeys{/tikz}{\pgfkeys@searchalso@name/.expand once=%
                  \pgfkeys@searchalso@temp@value}%
            \fi
        }{%
            \pgfkeysgetvalue{/handlers/.unknown/.@cmd}{\pgfkeys@code}%
            \expandafter\pgfkeys@code\pgfkeys@searchalso@temp@value\pgfeov
        }%
    }
}
```

2. `\pgfkeys{/path/.search also={/tikz,/pgf}}` is equivalent to

```
\pgfkeys{/path/.unknown/.code={%
        \def\pgfkeys@searchalso@temp@value{#1}%
        \ifpgfkeysaddeddefaultpath
            \expandafter\pgfkeys@firstoftwo
        \else
            \expandafter\pgfkeys@secondoftwo
        \fi{%
            \pgfkeyssuccessfalse
            \let\pgfkeys@searchalso@name=\pgfkeyscurrentkeyRAW
            \ifpgfkeyssuccess
            \else
                % step 1: search in /tikz with .try:
                \pgfqkeys{/tikz}{\pgfkeys@searchalso@name/.try/.expand once=%
                  \pgfkeys@searchalso@temp@value}%
            \fi
            \ifpgfkeyssuccess
            \else
                % step 2: search in /pgf (without .try!):
                \pgfqkeys{/pgf}{\pgfkeys@searchalso@name/.expand once=\pgfkeys@searchalso@}%
            \fi
        }{%
            \pgfkeysgetvalue{/handlers/.unknown/.@cmd}{\pgfkeys@code}%
            \expandafter\pgfkeys@code\pgfkeys@searchalso@temp@value\pgfeov
        }%
    }
}
```

To also enable searching for styles (or other handled keys), consider changing the configuration for handled keys to `/handler config=full or existing` when you use `/.search also`, that is, use

```
\pgfkeys{
   /main path/.search also={/secondary path},
   /handler config=full or existing}
```

87.4.8 Handlers for Testing Keys

Key handler ⟨*key*⟩`/.try`=⟨*value*⟩

This handler causes the same things to be done as if ⟨*key*⟩=⟨*value*⟩ had been written instead. However, if neither ⟨*key*⟩`/.@cmd` nor the key itself is defined, no handlers will be called. Instead, the execution of the key just stops. Thus, this handler will "try" to use the key, but no further action is taken when the key is not defined.

The TEX-if `\ifpgfkeyssuccess` will be set according to whether the ⟨*key*⟩ was successfully executed or not.

(a:hallo)(b:welt)

```
\pgfkeys{/a/.code=(a:#1)}
\pgfkeys{/b/.code=(b:#1)}
\pgfkeys{/x/.try=hmm,/a/.try=hallo,/b/.try=welt}
```

Key handler ⟨*key*⟩`/.retry`=⟨*value*⟩

This handler works just like `/.try`, only it will not do anything if `\ifpgfkeyssuccess` is false. Thus, this handler will only retry to set a key if "the last attempt failed".

(a:hallo)

```
\pgfkeys{/a/.code=(a:#1)}
\pgfkeys{/b/.code=(b:#1)}
\pgfkeys{/x/.try=hmm,/a/.retry=hallo,/b/.retry=welt}
```

Key handler ⟨*key*⟩`/.lastretry`=⟨*value*⟩

This handler works like `/.retry`, only it will invoke the usual handlers for unknowns keys if `\ifpgfkeyssuccess` is false. Thus, this handler will only try to set a key if "the last attempt failed". Furthermore, this here is the last such attempt.

87.4.9 Handlers for Key Inspection

Key handler ⟨*key*⟩`/.show value`

This handler executes a `\show` command on the value stored in ⟨*key*⟩. This is useful mostly for debugging.

Example: `\pgfkeys{/my/obscure key/.show value}`

Key handler ⟨*key*⟩`/.show code`

This handler executes a `\show` command on the code stored in ⟨*key*⟩`/.@cmd`. This is useful mostly for debugging.

Example: `\pgfkeys{/my/obscure key/.show code}`

The following key is not a handler, but it also commonly used for inspecting things:

`/utils/exec=`⟨*code*⟩ (no default)

This key will simply execute the given ⟨*code*⟩.

Example: `\pgfkeys{some key=some value,/utils/exec=\show\hallo,obscure key=obscure}`

87.5 Error Keys

In certain situations errors can occur, like using an undefined key. In these situations error keys are executed. They should store a macro that gets two arguments: The first is the offending key (possibly only after macro expansion), the second is the value that was passed as a parameter (also possibly only after macro expansion).

Currently, error keys are simply executed. In the future it might be a good idea to have different subkeys that are executed depending on the language currently set so that users get a localized error message.

`/errors/value required={`⟨*offending key*⟩`}{`⟨*value*⟩`}` (no default)

This key is executed whenever an ⟨*offending key*⟩ is used without a value when a value is actually required.

`/errors/value forbidden={`⟨*offending key*⟩`}{`⟨*value*⟩`}` (no default)

This key is executed whenever a key is used with a value when a value is actually forbidden.

`/errors/boolean expected={`⟨*offending key*⟩`}{`⟨*value*⟩`}` (no default)

This key is executed whenever a key set up using `/.is if` gets called with a ⟨*value*⟩ other than `true` or `false`.

`/errors/unknown choice value={`⟨*offending key*⟩`}{`⟨*value*⟩`}` (no default)

This key is executed whenever a choice is used as a ⟨*value*⟩ for a key set up using the `/.is choice` handler that is not defined.

`/errors/unknown key={`⟨*offending key*⟩`}{`⟨*value*⟩`}` (no default)

This key is executed whenever a key is unknown and no specific `/.unknown` handler is found.

87.6 Key Filtering

An extension by Christian Feuersänger

Normally, a call to `\pgfkeys` sets all keys provided in its argument list. This is usually what users expect it to do. However, implementations of different packages or PGF-libraries may need more control over the key setting procedure: library A may want to set its options directly and communicate all remaining ones to library B.

pgfkeys Library `filtered`

```
\usepgfkeyslibrary{filtered} % LaTeX and plain TeX
\usepgfkeyslibrary[filtered] % ConTeXt
```

The `filtered` library provides functionality to filter keys and only process selected ones, while forwarding the others to another namespace.

This library is loaded by default by `pgfkeys`.

This section describes key filtering methods of PGF, including options for family groupings. If you merely want to use PGF (or its libraries), you can skip this section. It is addressed to package (or library) authors.

87.6.1 Starting With An Example

Users of `xkeyval` are familiar with the concept of key families: keys belong to groups and those keys can be 'filtered' out of other options. PGF supports family groupings and more abstract key selection mechanism with `\pgfkeysfiltered`, a variant of `\pgfkeys`. Suppose we have the example key grouping

```
\pgfkeys{
    /my group/A1/.code=(A1:#1),
    /my group/A2/.code=(A2:#1),
    /my group/A3/.code=(A3:#1),
    /my group/B/.code=(B:#1),
    /my group/C/.code=(B:#1),
}
```

and we want to set options `A1`, `A2` and `A3` only. A call to `\pgfkeys` yields

(A1:a1)(A2:a2)(B:b)(B:c)

```
\pgfkeys{/my group/A1=a1, /my group/A2=a2, /my group/B=b,  /my group/C=c}
```

because all those command option are processed consecutively.

Now, let's define a family named `A` which contains `A1`, `A2` and `A3` and set only family members of `A`. We prepare our key settings with

```
\pgfkeys{
    /my group/A/.is family,
    /my group/A1/.belongs to family=/my group/A,
    /my group/A2/.belongs to family=/my group/A,
    /my group/A3/.belongs to family=/my group/A,
}
```

and

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/pgf/key filters/active families/.install key filter}
```

After this preparation, we can use `\pgfkeysfiltered` with

(A1:a1)(A2:a2)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/my group/A/.activate family}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
 /my group/B=b,  /my group/C=c}
```

or

(A1:a1)(A2:a2)(A3:a3)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/my group/A/.activate family}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
 /my group/B=b,  /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

to set only keys which belong to an 'active' family – in our case, only family `A` was active, so the remaining options have not been processed. The family processing is quite fast and allows an arbitrary number of active key families.

Unprocessed options can be collected into a macro (similar to `xkeyval`'s `\xkv@rm`), discarded or handled manually. The details of key selection and family declaration are described in the following sections.

87.6.2 Setting Filters

The command `\pgfkeysfiltered` is the main tool to process only selected options. It works as follows.

`\pgfkeysfiltered{`⟨*key–value-list*⟩`}`

Processes all options in exactly the same way as `\pgfkeys{`⟨*key–value-list*⟩`}`, but a key filter is considered as soon as key identification is complete.

The key filter tells `\pgfkeysfiltered` whether it should continue to apply the current option (return value is 'true') or whether something different shall be done (filter returns 'false').

There is exactly one key filter in effect, and it is installed by the `.install key filter` handler or by `\pgfkeysinstallkeyfilter`.

If the key filter returns 'false', a unique key filter handler gets control. This handler is installed by the `.install key filter handler` method and has access to the key's full name, value and (possibly) path.

Key filtering applies to any (possibly nested) call to `\pgfkeys`, `\pgfkeysalso`, `\pgfqkeys` and `\pgfqkeysalso` during the evaluation of {⟨*key–value-list*⟩}. It does *not* apply to routines like `\pgfkeyssetvalue` or `\pgfkeysgetvalue`. Furthermore, keys belonging to `/errors` are always processed. Key filtering routines can't be nested: you can't combine different key filters automatically.

`\pgfqkeysfiltered`{⟨*default-path*⟩}{⟨*key–value-list*⟩}

A variant of `\pgfkeysfiltered` which uses the 'quick' search path setting. It is the `\pgfqkeys` variant of `\pgfkeysfiltered`, see the documentation for `\pgfqkeys` for more details.

`\pgfkeysalsofrom`{⟨*macro*⟩}

A variant of `\pgfkeysalso` which loads its key list from {⟨*macro*⟩}.

It is useful in conjunction with the `/pgf/key filter handlers/append filtered to=`⟨*macro*⟩ handler.

The following example uses the same settings as in the intro section 87.6.1.

(A1:a1)(A2:a2)(A3:a3)Remaining: '/my group/B=b,/my group/C=c,/tikz/color=blue'.(B:b)(B:c)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
 /my group/B=b,  /my group/C=c, /tikz/color=blue, /my group/A3=a3}

Remaining: `\remainingoptions'.
\pgfkeysalsofrom{\remainingoptions}
```

`\pgfkeysalsofiltered`{⟨*key–value-list*⟩}

This command works as `\pgfkeysfiltered`, but it does not change the current default path. See the documentation of `\pgfkeysalso` for more details.

`\pgfkeysalsofilteredfrom`{⟨*macro*⟩}

A variant of `\pgfkeysalsofiltered` which loads its key list from {⟨*macro*⟩}.

Key handler ⟨*key*⟩`/.install key filter=`⟨*optional arguments*⟩

This handler installs a key filter. A key filter is a command key which sets the TEX-boolean `\ifpgfkeysfiltercontinue`, that means a key with existing '`/.@cmd`' suffix. A simple example is a key filter which returns always true:

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/foo/bar/true key filter/.code={\pgfkeysfiltercontinuetrue}}
\pgfkeys{/foo/bar/true key filter/.install key filter}
```

If key filters require arguments, they are installed by `.install key filter` as well. An example is the `/pgf/key filters/equals` handler:

(A1:a1)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/pgf/key filters/equals/.install key filter={/my group/A1}}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
/my group/B=b,  /my group/C=c, /tikz/color=blue, /my group/A3=a3}
```

If a key filter requires more than one argument, you need to provide the complete argument list in braces like `{{first}{second}}`.

You can also use `\pgfkeysinstallkeyfilter`⟨*full key*⟩⟨*optional arguments*⟩, it has the same effect.

See section 87.6.7 for how to write key filters.

Key handler ⟨*key*⟩`/.install key filter handler=`⟨*optional arguments*⟩

This handler installs the routine which will be invoked for every *unprocessed* option, that means any option for which the key filter returned 'false'.

The `.install key filter handler` is used in the same way as `.install key filter`. There exists a macro version, `\pgfkeysinstallkeyfilterhandler`⟨*full key*⟩⟨*optional arguments*⟩, which has the same effect.

See section 87.6.7 for how to write key filter handlers.

87.6.3 Handlers For Unprocessed Keys

Each option for which key filters decided to skip them is handed over to a 'key filter handler'. There are several predefined key filter handlers.

`/pgf/key filter handlers/append filtered to={`⟨*macro*⟩`}` (no default)

Install this filter handler to append any unprocessed options to macro `{`⟨*macro*⟩`}`.

(A1:a1)(A2:a2)Remaining options: '/my group/B=b,/my group/C=c,/tikz/color=blue'.

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/pgf/key filter handlers/append filtered to/.install key filter handler=\remainingoptions}
\def\remainingoptions{}
\pgfkeysfiltered{/my group/A1=a1, /my group/A2=a2,
 /my group/B=b,  /my group/C=c, /tikz/color=blue}

Remaining options: `\remainingoptions'.
```

This example uses the same keys as defined in the intro section 87.6.1.

`/pgf/key filter handlers/ignore` (no value)

Install this filter handler if you simply want to ignore any unprocessed option. This is the default.

`/pgf/key filter handlers/log` (no value)

This key filter handler writes messages for any unprocessed option to your logfile (and terminal).

87.6.4 Family Support

PGF supports a family concept: every option can be associated with (at most) one family. Families form loose key groups which are independent of the key hierarchy. For example, `/my tree/key1` can belong to family `/tikz`.

It is possible to 'activate' or 'deactivate' single families. Furthermore, it is possible to set only keys which belong to active families using appropriate key filter handlers.

The family support is fast: if there are N options in a key–value-list and there are K active families, the runtime for `\pgfkeysfiltered` is $O(N + K)$ (activate every family $O(K)$, check every option $O(N)$, deactivate every family $O(K)$).

Key handler ⟨*key*⟩`/.is family`

Defines a new family. This option has already been described in section 87.4.1 on page 986.

Key handler ⟨*key*⟩`/.activate family`

Activates a family. The family needs to be defined, otherwise `/errors/family unknown` will be raised.

Activation means a TEX-boolean will be set to `true`, indicating that a family should be processed.

You can also use `\pgfkeysactivatefamily`⟨*full path*⟩ to get the same effect. Furthermore, you can use `\pgfkeysactivatefamilies`⟨*list of families*⟩⟨*macro name for de-activation*⟩ to activate a list of families (see section 87.6.6).

Key handler ⟨*key*⟩`/.deactivate family`

Deactivates a family. The family needs to be defined, otherwise `/errors/family unknown` will be raised.

You can also use `\pgfkeysdeactivatefamily`⟨*full path*⟩ to get the same effect.

Key handler ⟨*key*⟩`/.belongs to family={`⟨*family name*⟩`}`

Associates the current option with `{`⟨*family name*⟩`}`, which is expected to be a full path of a family.

```
\usepgfkeyslibrary {filtered}
\pgfkeys{/foo/bar/.is family}
\pgfkeys{
    /foo/a/.belongs to family=/foo/bar,
    /foo/b/.belongs to family=/foo/bar
}
```

Each option can have up to one family, `.belongs to family` overwrites any old setting.

/pgf/key filters/active families (no value)

Install this key filter if `\pgfkeysfiltered` should only process activated families. If a key does not belong to any family, it is not processed. If a key is completely unknown within the default path, the normal 'unknown' handlers of `\pgfkeys` are invoked.

/pgf/key filters/active families or no family={⟨*key filter 1*⟩}{⟨*key filter 2*⟩} (no default)

This key filter configures `\pgfkeysfiltered` to work as follows.

1. If the current key belongs to a family, set `\ifpgfkeysfiltercontinue` to true if and only if its family is active.
2. If the current key does *not* belong to a family, assign `\ifpgfkeysfiltercontinue` as result of {⟨*key filter 1*⟩}.
3. If the current key is unknown within the default path, assign `\ifpgfkeysfiltercontinue` as result of {⟨*key filter 2*⟩}.

The arguments {⟨*key filter 1*⟩} and {⟨*key filter 2*⟩} are other key filters (possibly with options) and allow fine-grained control over the filtering process.

```
\usepgfkeyslibrary {filtered}
\pgfkeysinstallkeyfilter
    {/pgf/key filters/active families or no family}
    {{/pgf/key filters/is descendant of=/tikz}% for keys without family
     {/pgf/key filters/false}% for unknown keys
    }%
```

This key filter will return true for any option with active family. If an option has no family, the return value is true if and only if it belongs to `/tikz`. If the option is unknown, the return value is `false` and unknown handlers won't be called.

/pgf/key filters/active families or no family DEBUG={⟨*key filter 1*⟩}{⟨*key filter 2*⟩} (no default)

A variant of `active families or no family` which protocols each action on your terminal (log-file).

/pgf/key filters/active families and known (no value)

A fast alias for

```
/pgf/key filters/active families or no family=
  {/pgf/keys filters/false}
  {/pgf/keys filters/false}.
```

/pgf/key filters/active families or descendants of={⟨*path prefix*⟩} (no default)

A fast alias for

```
/pgf/key filters/active families or no family=
  {/pgf/keys filters/is descendant of={⟨path prefix⟩}}
  {/pgf/keys filters/false}.
```

\pgfkeysactivatefamiliesandfilteroptions{⟨*family list*⟩}{⟨*key–value-list*⟩}

A simple shortcut macro which activates any family in the comma separated {⟨*family list*⟩}, invokes `\pgfkeysfiltered`⟨*key–value-list*⟩ and deactivates the families afterwards.

Please note that you will need to install a family key filter, otherwise family activation has no effect.

\pgfqkeysactivatefamiliesandfilteroptions{⟨*family list*⟩}{⟨*default path*⟩}{⟨*key–value-list*⟩}

The 'quick' default path variant of `\pgfkeysactivatefamiliesandfilteroptions`.

\pgfkeysactivatesinglefamilyandfilteroptions{⟨*family name*⟩}{⟨*key–value-list*⟩}

A shortcut macro which activates a single family and invokes `\pgfkeysfiltered`.

Please note that you will need to install a family key filter, otherwise family activation has no effect.

\pgfqkeysactivatesinglefamilyandfilteroptions{⟨*family name*⟩}{⟨*default path*⟩}{⟨*key–value-list*⟩}

The 'quick' default path variant of `\pgfkeysactivatesinglefamilyandfilteroptions`.

87.6.5 Other Key Filters

There are some more key filters which have nothing to do with family handling.

`/pgf/key filters/is descendant of=`{⟨*path*⟩} (no default)

Install this key filter to process only options belonging to the key tree ⟨*path*⟩. It returns true for every key whose key path is equal to ⟨*path*⟩. It also returns true for any unknown key, that means unknown keys are processed using the standard unknown handlers of PGF.

(A:a)(B:b)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{
/group 1/A/.code={(A:#1)},
/group 1/foo/bar/B/.code={(B:#1)},
/group 2/C/.code={(C:#1)},
/pgf/key filters/is descendant of/.install key filter=/group 1}
\pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

`/pgf/key filters/equals=`{⟨*full key*⟩} (no default)

Install this key filter to process only the fully qualified option {⟨*full key*⟩}. The filter returns true for any unknown key or if the key equals {⟨*full key*⟩}.

(A:a)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{
/group 1/A/.code={(A:#1)},
/group 1/B/.code={(B:#1)},
/pgf/key filters/equals/.install key filter=/group 1/A}
\pgfqkeysfiltered{/group 1}{A=a,B=b}
```

`/pgf/key filters/not=`{⟨*key filter*⟩} (no default)

This key filter logically inverts the result of {⟨*key filter*⟩}.

(C:c)

```
\usepgfkeyslibrary {filtered}
\pgfkeys{
/group 1/A/.code={(A:#1)},
/group 1/foo/bar/B/.code={(B:#1)},
/group 2/C/.code={(C:#1)},
/pgf/key filters/not/.install key filter=
    {/pgf/key filters/is descendant of=/group 1}}
\pgfkeysfiltered{/group 1/A=a,/group 1/foo/bar/B=b,/group 2/C=c}
```

Please note that unknown keys will be handed to the usual unknown handlers.

`/pgf/key filters/and=`{⟨*key filter 1*⟩}{⟨*key filter 2*⟩} (no default)

This key filter returns true if and only if both, {⟨*key filter 1*⟩} and {⟨*key filter 2*⟩} return true.

`/pgf/key filters/or=`{⟨*key filter 1*⟩}{⟨*key filter 2*⟩} (no default)

This key filter returns true if one of {⟨*key filter 1*⟩} and {⟨*key filter 2*⟩} returns true.

`/pgf/key filters/true` (no value)

This key filter returns always true.

`/pgf/key filters/false` (no value)

This key filter returns always false (including unknown keys).

`/pgf/key filters/defined` (no value)

This key filter returns false if the current key is unknown, which avoids calling the unknown handlers.

87.6.6 Programmer Interface

`\pgfkeysinterruptkeyfilter`
⟨*environment contents*⟩

\endpgfkeysinterruptkeyfilter

Temporarily disables key filtering inside the environment. If key filtering is not active, this has no effect at all.

Please note that no TEX-group is introduced.

\pgfkeyssavekeyfilterstateto{⟨*macro*⟩}

Creates {⟨*macro*⟩} which contains commands to re-activate the current key filter and key filter handler. It can be used to temporarily switch the key filter.

\pgfkeysinstallkeyfilter{⟨*full key*⟩}{⟨*optional arguments*⟩}

The command \pgfkeysinstallkeyfilter{⟨*full key*⟩}{⟨*optional arguments*⟩} has the same effect as \pgfkeys{⟨*full key*⟩/.install key filter={⟨*optional arguments*⟩}}.

\pgfkeysinstallkeyfilterhandler{⟨*full key*⟩}{⟨*optional arguments*⟩}

The command \pgfkeysinstallkeyfilterhandler{⟨*full key*⟩}{⟨*optional arguments*⟩} has the same effect as \pgfkeys{⟨*full key*⟩/.install key filter handler={⟨*optional arguments*⟩}}.

\pgfkeysactivatefamily{⟨*family name*⟩}

Equivalent to \pgfkeys{⟨*family name*⟩/.activate family}.

\pgfkeysdeactivatefamily{⟨*family name*⟩}

Equivalent to \pgfkeys{⟨*family name*⟩/.deactivate family}.

\pgfkeysactivatefamilies{⟨*family list*⟩}{⟨*deactivate macro name*⟩}

Activates each family in ⟨*family list*⟩ and creates a macro ⟨*deactivate macro name*⟩ which deactivates each family in ⟨*family list*⟩.

```
\usepgfkeyslibrary {filtered}
\pgfkeysactivatefamilies{/family 1,/family 2,/family 3}{\deactivatename}
\pgfkeysfiltered{foo,bar}
\deactivatename
```

\pgfkeysiffamilydefined{⟨*family*⟩}{⟨*true case*⟩}{⟨*false case*⟩}

Checks whether the full key ⟨*family*⟩ is a family and executes either ⟨*true case*⟩ or ⟨*false case*⟩.

\pgfkeysisfamilyactive{⟨*family*⟩}

Sets the TEX-boolean \ifpgfkeysfiltercontinue to whether ⟨*family*⟩ is active or not.

\pgfkeysgetfamily{⟨*full key*⟩}{⟨*resultmacro*⟩}

Returns the family associated to a ⟨*full key*⟩ into macro ⟨*resultmacro*⟩.

\pgfkeyssetfamily{⟨*full key*⟩}{⟨*family*⟩}

The command \pgfkeyssetfamily{⟨*full key*⟩}{⟨*family*⟩} has the same effect as \pgfkeys{⟨*full key*⟩/.belongs to family={⟨*family*⟩}}.

87.6.7 Defining Own Filters Or Filter Handlers

During \pgfkeysfiltered, the key filter code will be invoked. At this time, the full key path including key name is available as \pgfkeyscurrentkey, the key name before default paths have been considered as \pgfkeyscurrentkeyRAW and the values as \pgfkeyscurrentvalue.

Furthermore, the macro \pgfkeyscasenumber contains the current key's type as an integer:

⟨*1*⟩ The key is a command key (i.e. .../.@cmd exists).

⟨*2*⟩ The key contains its value directly.

⟨*3*⟩ The key is handled (for example it is .code or .cd).

In this case, the macros \pgfkeyscurrentname and \pgfkeyscurrentpath are set to the handlers name and path, respectively. Invoke \pgfkeyssplitpath{} to extract these values for non-handled keys.

⟨*0*⟩ The key is unknown.

Any key filter or key filter handler can access these variables. Key filters are expected to set the TEX-boolean `\ifpgfkeysfiltercontinue` to whether the current key shall be processed or not.

`\pgfkeysevalkeyfilterwith{`⟨*full key*⟩`}={`⟨*filter arguments*⟩`}`

Evaluates a fully qualified key filter ⟨*full key*⟩ with argument(s) ⟨*filter arguments*⟩.

```
\usepgfkeyslibrary {filtered}
\pgfkeysevalkeyfilterwith{/pgf/key filters/equals=/tikz}
```

88 Repeating Things: The Foreach Statement

This section describes the package `pgffor`, which is loaded automatically by Ti*k*Z, but not by PGF:

```
\usepackage{pgffor} % LaTeX
\input{pgffor.tex} % plain TeX
\usemodule[pgffor]  % ConTeXt
```

This package can be used independently of PGF, but works particularly well together with PGF and Ti*k*Z. It defines two new commands: `\foreach` and `\breakforeach`.

`\foreach` ⟨*variables*⟩ `[`⟨*options*⟩`]` `in` ⟨*list*⟩ ⟨*commands*⟩

The syntax of this command is a bit complicated, so let us go through it step-by-step.

In the easiest case, ⟨*variables*⟩ is a single TeX-command like `\x` or `\point`. (If you want to have some fun, you can also use active characters. If you do not know what active characters are, you are blessed.)

Still in the easiest case, ⟨*options*⟩ will be omitted. The keys for customizing this command will be discussed below.

Again, in the easiest case, ⟨*list*⟩ is either a comma-separated list of values surrounded by curly braces or it is the name of a macro that contain such a list of values. Anything can be used as a value, but numbers are most likely.

Finally, in the easiest case, ⟨*commands*⟩ is some TeX-text in curly braces.

With all these assumptions, the `\foreach` statement will execute the ⟨*commands*⟩ repeatedly, once for every element of the ⟨*list*⟩. Each time the ⟨*commands*⟩ are executed, the ⟨*variable*⟩ will be set to the current value of the list item.

[1][2][3][0]

```
\foreach \x in {1,2,3,0} {[\x]}
```

[1][2][3][0]

```
\def\mylist{1,2,3,0}
\foreach \x in \mylist {[\x]}
```

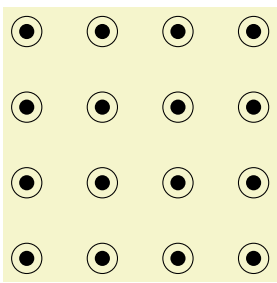
Note that in each execution of ⟨*commands*⟩ the ⟨*commands*⟩ are put in a TeX group. This means that *local changes to counters inside* ⟨*commands*⟩ *do not persist till the next iteration.* For instance, if you add 1 to a counter inside ⟨*commands*⟩ locally, then in the next iteration the counter will have the same value it had at the beginning of the first iteration. You have to add `\global` if you wish changes to persist from iteration to iteration.

Syntax for the commands. Let us move on to a more complicated setting. The first complication occurs when the ⟨*commands*⟩ are not some text in curly braces. If the `\foreach` statement does not encounter an opening brace, it will instead scan everything up to the next semicolon and use this as ⟨*commands*⟩. This is most useful in situations like the following:



```
\tikz
  \foreach \x in {0,1,2,3}
    \draw (\x,0) circle [radius=0.2cm];
```

However, the "reading till the next semicolon" is not the whole truth. There is another rule: If a `\foreach` statement is directly followed by another `\foreach` statement, this second foreach statement is collected as ⟨*commands*⟩. This allows you to write the following:



```
\begin{tikzpicture}
  \foreach \x in {0,1,2,3}
    \foreach \y in {0,1,2,3}
      {
        \draw (\x,\y) circle [radius=0.2cm];
        \fill (\x,\y) circle [radius=0.1cm];
      }
\end{tikzpicture}
```

The dots notation. The second complication concerns the ⟨*list*⟩. If this ⟨*list*⟩ contains the list item "`...`", this list item is replaced by the "missing values". More precisely, the following happens:

Normally, when a list item `...` is encountered, there should already have been *two* list items before it, which were numbers. Examples of *numbers* are `1`, `-10`, or `-0.24`. Let us call these numbers x and y and let $d := y - x$ be their difference. Next, there should also be one number following the three dots, let us call this number z.

In this situation, the part of the list reading "x`,`y`,...,`z" is replaced by "x, $x + d$, $x + 2d$, $x + 3d$, ..., $x + md$", where the last dots are semantic dots, not syntactic dots. The value m is the largest number such that $x + md \leq z$ if d is positive or such that $x + md \geq z$ if d is negative.

Perhaps it is best to explain this by some examples: The following ⟨*list*⟩ have the same effects:

`\foreach \x in {1,2,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,2,3,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {1,3,...,11} {\x, }` yields 1, 3, 5, 7, 9, 11,

`\foreach \x in {1,3,...,10} {\x, }` yields 1, 3, 5, 7, 9,

`\foreach \x in {0,0.1,...,0.5} {\x, }` yields 0, 0.1, 0.20001, 0.30002, 0.40002,

`\foreach \x in {a,b,9,8,...,1,2,2.125,...,2.5} {\x, }` yields a, b, 9, 8, 7, 6, 5, 4, 3, 2, 1, 2, 2.125, 2.25, 2.375, 2.5,

As can be seen, for fractional steps that are not multiples of 2^{-n} for some small n, rounding errors can occur pretty easily. Thus, in the second last case, `0.5` should probably be replaced by `0.501` for robustness.

There is another special case for the `...` statement: If the `...` is used right after the first item in the list, that is, if there is an x, but no y, the difference d obviously cannot be computed and is set to 1 if the number z following the dots is larger than x and is set to -1 if z is smaller:

`\foreach \x in {1,...,6} {\x, }` yields 1, 2, 3, 4, 5, 6,

`\foreach \x in {9,...,3.5} {\x, }` yields 9, 8, 7, 6, 5, 4,

There is a yet another special case for the `...` statement, in that it can indicate an alphabetic character sequence:

`\foreach \x in {a,...,m} {\x, }` yields a, b, c, d, e, f, g, h, i, j, k, l, m,

`\foreach \x in {Z,X,...,M} {\x, }` yields Z, X, V, T, R, P, N,

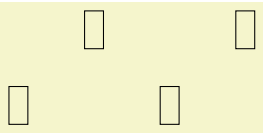
A final special case for the `...` statement is contextual replacement. If the `...` is used in some context, for example, `sin(...)`, this context will be interpreted correctly, provided that the list items prior to the `...` statement have *exactly* the same pattern, except that, instead of dots, they have a number or a character:

`\foreach \x in {2^1,2^...,2^7} {$\x$, }` yields 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7,

`\foreach \x in {0\pi,0.5\pi,...\pi,3\pi} {$\x$, }` yields 0π, 0.5π, 1π, 1.5π, 2π, 2.5π, 3π,

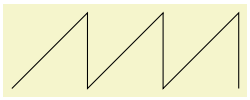
`\foreach \x in {A_1,..._1,H_1} {$\x$, }` yields A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1,

Special handling of pairs. Different list items are separated by commas. However, this causes a problem when the list items contain commas themselves as pairs like `(0,1)` do. In this case, you should put the items containing commas in braces as in `{(0,1)}`. However, since pairs are such a natural and useful case, they get a special treatment by the `\foreach` statement. When a list item starts with a `(` everything up to the next `)` is made part of the item. Thus, we can write things like the following:

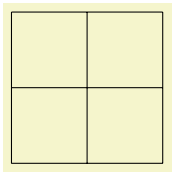


```
\tikz
  \foreach \position in {(0,0), (1,1), (2,0), (3,1)}
    \draw \position rectangle +(.25,.5);
```

Using the foreach-statement inside paths. Ti*k*Z allows you to use `foreach` and `\foreach` (both have the same effect) inside a path construction. In such a case, the ⟨*commands*⟩ must be path construction commands. Here are two examples:



```
\tikz
  \draw (0,0)
    foreach \x in {1,...,3}
      { -- (\x,1) -- (\x,0) }
    ;
```



```
\tikz \draw foreach \p in {1,...,3} {(\p,1)--(\p,3) (1,\p)--(3,\p)};
```

Note that the `node` and `pic` path commands also support the `foreach` statement in special ways.

Multiple variables. You will often wish to iterate over two variables at the same time. Since you can nest `\foreach` loops, this is normally straight-forward. However, you sometimes wish variables to iterate "simultaneously". For example, we might be given a list of edges that connect two coordinates and might wish to iterate over these edges. While doing so, we would like the source and target of the edges to be set to two different variables.

To achieve this, you can use the following syntax: The ⟨*variables*⟩ may not only be a single TEX-variable. Instead, it can also be a list of variables separated by slashes (`/`). In this case the list items can also be lists of values separated by slashes.

Assuming that the ⟨*variables*⟩ and the list items are lists of values, each time the ⟨*commands*⟩ are executed, each of the variables in ⟨*variables*⟩ is set to one part of the list making up the current list item. Here is an example to clarify this:

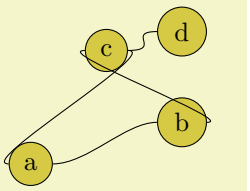
Example: `\foreach \x / \y in {1/2,a/b} {``\x\ and \y''}` yields "1 and 2""a and b".

If some entry in the ⟨*list*⟩ does not have "enough" slashes, the last entry will be repeated. Here is an example:

0 1 2 *e* 3

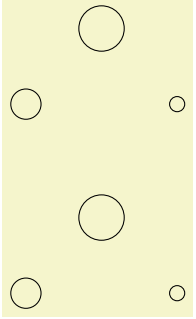
```
\begin{tikzpicture}
  \foreach \x/\xtext in {0,...,3,2.72 / e}
    \draw (\x,0) node{$\xtext$};
\end{tikzpicture}
```

Here are more useful examples:



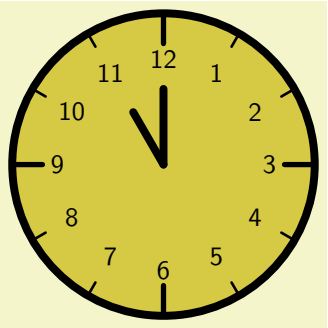

```
\begin{tikzpicture}
  % Define some coordinates:
  \path[nodes={circle,fill=yellow!80!black,draw}]
    (0,0)    node(a) {a}
    (2,0.55) node(b) {b}
    (1,1.5)  node(c) {c}
    (2,1.75) node(d) {d};

  % Draw some connections:
  \foreach \source/\target in {a/b, b/c, c/a, c/d}
    \draw (\source) .. controls +(.75cm,0pt) and +(-.75cm,0pt)..(\target);
\end{tikzpicture}
```



```
\begin{tikzpicture}
  % Let's draw circles at interesting points:
  \foreach \x / \y / \r in {0 / 0 / 2mm, 1 / 1 / 3mm, 2 / 0 / 1mm}
    \draw (\x,\y) circle [radius=\r];

  % Same effect
  \foreach \center/\r in {{(0,0)/2mm}, {(1,1)/3mm}, {(2,0)/1mm}}
    \draw[yshift=2.5cm] \center circle [radius=\r];
\end{tikzpicture}
```

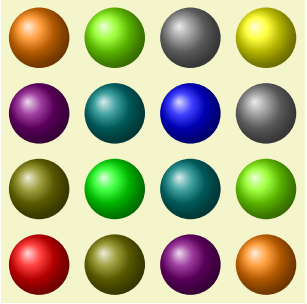



```
\begin{tikzpicture}[line cap=round,line width=3pt]
  \filldraw [fill=yellow!80!black] (0,0) circle [radius=2cm];

  \foreach \angle / \label in
    {0/3, 30/2, 60/1, 90/12, 120/11, 150/10, 180/9,
     210/8, 240/7, 270/6, 300/5, 330/4}
  {
    \draw[line width=1pt] (\angle:1.8cm) -- (\angle:2cm);
    \draw (\angle:1.4cm) node{\textsf{\label}};
  }

  \foreach \angle in {0,90,180,270}
    \draw[line width=2pt] (\angle:1.6cm) -- (\angle:2cm);

  \draw (0,0) -- (120:0.8cm); % hour
  \draw (0,0) -- (90:1cm);    % minute
\end{tikzpicture}%
```



```
\tikz[shading=ball]
  \foreach \x / \cola in {0/red,1/green,2/blue,3/yellow}
    \foreach \y / \colb in {0/red,1/green,2/blue,3/yellow}
      \shade[ball color=\cola!50!\colb] (\x,\y) circle [radius=0.4cm];
```

Options to customize the foreach-statement.

The keys described below can be used in the ⟨*options*⟩ argument to the `\foreach` command. They all have the path `/pgf/foreach/`, however, the path is set automatically when ⟨*options*⟩ are parsed, so it does not have to be explicitly stated.

`/pgf/foreach/var=`⟨*variable*⟩ (no default)

This key provides an alternative way to specify variables: `\foreach [var=\x,var=\y]` is the same as `\foreach \x/\y`. If used, this key should be used before the other keys.

`/pgf/foreach/evaluate=`⟨*variable*⟩ `as` ⟨*macro*⟩ `using` ⟨*formula*⟩ (no default)

By default, list items are not evaluated: `1+2`, yields `1+2`, not `3`. This key allows a variable to be evaluated using the mathematical engine. The variable must have been specified either using the `var` key or in the ⟨*variables*⟩ argument of the `foreach` command. By default, the result of the evaluation will be stored in ⟨*variable*⟩. However, the optional `as` ⟨*macro*⟩ statement can be used to store the result in ⟨*macro*⟩.

1.0, 2.0, 4.0, 8.0, 16.0, 32.0, 64.0, 128.0, 256.0,

```
\foreach \x [evaluate=\x] in {2^0,2^...,2^8}{$\x$, }
```

$2^0 = 1.0$, $2^1 = 2.0$, $2^2 = 4.0$, $2^3 = 8.0$, $2^4 = 16.0$, $2^5 = 32.0$, $2^6 = 64.0$, $2^7 = 128.0$, $2^8 = 256.0$,

```
\foreach \x [evaluate=\x as \xeval] in {2^0,2^...,2^8}{$\x=\xeval$, }
```

The optional `using` ⟨*formula*⟩ statement means an evaluation does not have to be explicitly stated for each item in ⟨*list*⟩. The ⟨*formula*⟩ should contain at least one reference to ⟨*variable*⟩.

0 1 2 3 4 5 6 7 8 9 10

```
\tikz\foreach \x [evaluate=\x as \shade using \x*10] in {0,1,...,10}
  \node [fill=red!\shade!yellow, minimum size=0.65cm] at (\x,0) {\x};
```

`/pgf/foreach/remember=`⟨*variable*⟩ `as` ⟨*macro*⟩ `(initially` ⟨*value*⟩`)` (no default)

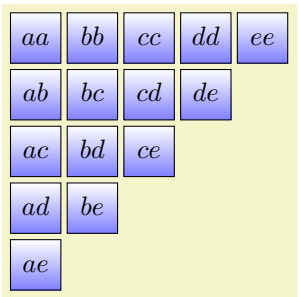
This key allows the item value stored in ⟨*variable*⟩ to be remembered during the next iteration, stored in ⟨*macro*⟩. If a variable is evaluated, the result of this evaluation is remembered. By default the value of ⟨*variable*⟩ is zero for the first iteration, however, the optional `(initially` ⟨*value*⟩`)` statement, allows the ⟨*macro*⟩ to be initially defined as ⟨*value*⟩.

$\overrightarrow{AB}$, $\overrightarrow{BC}$, $\overrightarrow{CD}$, $\overrightarrow{DE}$, $\overrightarrow{EF}$, $\overrightarrow{FG}$, $\overrightarrow{GH}$,

```
\foreach \x [remember=\x as \lastx (initially A)] in {B,...,H}{$\overrightarrow{\lastx\x}$, }
```

`/pgf/foreach/count=`⟨*macro*⟩ `from` ⟨*value*⟩ (no default)

This key allows ⟨*macro*⟩ to hold the position in the list of the current item. The optional `from` ⟨*value*⟩ statement allows the counting to begin from ⟨*value*⟩.




```
\tikz[x=0.75cm,y=0.75cm]
  \foreach \x [count=\xi] in {a,...,e}
    \foreach \y [count=\yi] in {\x,...,e}
      \node [draw, top color=white, bottom color=blue!50, minimum size=0.666cm]
        at (\xi,-\yi) {$\mathstrut\x\y$};
```

`/pgf/foreach/parse=`{⟨*boolean*⟩} (default `false`)

If this key is set to true the upper bound in the loop will be fed into `\pgfmathparse`. This allows to use complex expressions as the upper bound. However, the expression must be safe for evaluation in `\pgfmathparse`. It is known that internal TEX registers can cause trouble.

1 2 3 4 5 6 7 8 9

```
\foreach \x [parse=true] in {1,...,1.0e+1 - 1}{ \x }
```

`/pgf/foreach/expand list=`{⟨*boolean*⟩} (default `false`)

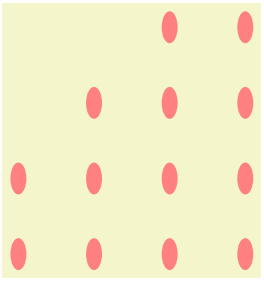
If this key is set to true the contents of the list are fully expanded with `\edef` before further processing. This allows using complex macros which generate a list upon expansion without having to use an intermediate macro.

1 2 3 4 5

```
\def\Iota#1#2{%
  \ifnum\numexpr#1\relax<\numexpr#2\relax
    \the\numexpr#1\relax,%
    \expandafter\Iota\expandafter{\the\numexpr(#1)+1\relax}{#2}%
  \else
    \the\numexpr#2\relax
  \fi}
\foreach [expand list=true] \x in {\Iota{1}{5}} {
  \x
}
```

\breakforeach

If this command is given inside a **\foreach** command, no further executions of the ⟨*commands*⟩ will occur. However, the current execution of the ⟨*commands*⟩ is continued normally, so it is probably best to use this command only at the end of a **\foreach** command.



```
\begin{tikzpicture}
  \foreach \x in {1,...,4}
    \foreach \y in {1,...,4}
    {
      \fill[red!50] (\x,\y) ellipse [x radius=3pt, y radius=6pt];

      \ifnum \x<\y
        \breakforeach
      \fi
    }
\end{tikzpicture}
```

89 Date and Calendar Utility Macros

This section describes the package `pgfcalendar`.

```
\usepackage{pgfcalendar} % LaTeX
\input{pgfcalendar.tex} % plain TeX
\usemodule[pgfcalendar]  % ConTeXt
```

This package can be used independently of PGF. It has two purposes:

1. It provides functions for working with dates. Most noticeably, it can convert a date in ISO-standard format (like 1975-12-26) to a so-called Julian day number, which is defined in Wikipedia as follows: "The Julian day or Julian day number is the (integer) number of days that have elapsed since the initial epoch at noon Universal Time (UT) Monday, January 1, 4713 BC in the proleptic Julian calendar". The package also provides a function for converting a Julian day number to an ISO-format date.

   Julian day numbers make it very easy to work with days. For example, the date ten days in the future of 2008-02-20 can be computed by converting this date to a Julian day number, adding 10, and then converting it back. Also, the day of week of a given date can be computed by taking the Julian day number modulo 7.

2. It provides a macro for typesetting a calendar. This macro is highly configurable and flexible (for example, it can produce both plain text calendars and also complicated Ti*k*Z-based calendars), but most users will not use the macro directly. It is the job of a frontend to provide useful configurations for typesetting calendars based on this command.

89.1 Handling Dates

89.1.1 Conversions Between Date Types

`\pgfcalendardatetojulian{`⟨*date*⟩`}{`⟨*counter*⟩`}`

This macro converts a date in a format to be described in a moment to the Julian day number in the Gregorian calendar. The ⟨*date*⟩ should expand to a string of the following form:

1. It should start with a number representing the year. Use `\year` for the current year, that is, the year the file is being typeset.
2. The year must be followed by a hyphen.
3. Next should come a number representing the month. Use `\month` for the current month. You can, but need not, use leading zeros. For example, `02` represents February, just like `2`.
4. The month must also be followed by a hyphen.
5. Next you must either provide a day of month (again, a number and, again, `\day` yields the current day of month) or the keyword `last`. This keyword refers to the last day of the month, which is automatically computed (and which is a bit tricky to compute, especially for February).
6. Optionally, you can next provide a plus sign followed by positive or negative number. This number of days will be added to the computed date.

Here are some examples:

- `2006-01-01` refers to the first day of 2006.
- `2006-02-last` refers to February 28, 2006.
- `\year-\month-\day` refers to today.
- `2006-01-01+2` refers to January 3, 2006.
- `\year-\month-\day+1` refers to tomorrow.
- `\year-\month-\day+-1` refers to yesterday.

The conversion method is taken from the English Wikipedia entry on Julian days.

Example: `\pgfcalendardatetojulian{2007-01-14}{\mycount}` sets `\mycount` to 2454115.

\pgfcalendarjuliantodate{⟨*Julian day*⟩}{⟨*year macro*⟩}{⟨*month macro*⟩}{⟨*day macro*⟩}

This command converts a Julian day number to an ISO-date. The ⟨*Julian day*⟩ must be a number or TEX counter, the ⟨*year macro*⟩, ⟨*month macro*⟩ and ⟨*day macro*⟩ must be TEX macro names. They will be set to numbers representing the year, month, and day of the given Julian day in the Gregorian calendar.

The ⟨*year macro*⟩ will be assigned the year without leading zeros. Note that this macro will produce year 0 (as opposed to other calendars, where year 0 does not exist). However, if you really need calendars for before the year 1, it is expected that you know what you are doing anyway.

The ⟨*month macro*⟩ gets assigned a two-digit number representing the month (with a leading zero, if necessary). Thus, the macro is set to `01` for January.

The ⟨*day macro*⟩ gets assigned a two-digit number representing the day of the month (again, possibly with a leading zero).

To convert a Julian day number to an ISO-date you use code like the following:

```
\pgfcalendarjuliantodate{2454115}{\myyear}{\mymonth}{\myday}
\edef\isodate{\myyear-\mymonth-\myday}
```

The above code sets `\isodate` to `2007-01-14`.

\pgfcalendarjuliantoweekday{⟨*Julian day*⟩}{⟨*week day counter*⟩}

This command converts a Julian day to a week day by computing the day modulo 7. The ⟨*week day counter*⟩ must be a TEX counter. It will be set to 0 for a Monday, to 1 for a Tuesday, and so on.

Example: `\pgfcalendarjuliantoweekday{2454115}{\mycount}` sets `\mycount` to 6 (it was a Sunday).

\pgfcalendareastersunday{⟨*year*⟩}{⟨*counter*⟩}

This command computes the date of Easter Sunday as a Julian date and stores it in ⟨*counter*⟩.

Example: `\pgfcalendareastersunday{2019}{\mycount}` sets `\mycount` to 2458595, which corresponds to `2019-04-21`.

89.1.2 Checking Dates

\pgfcalendarifdate{⟨*date*⟩}{⟨*tests*⟩}{⟨*code*⟩}{⟨*else code*⟩}

This command is used to execute code based on properties of ⟨*date*⟩. The ⟨*date*⟩ must be a date in ISO-format. For this date, the ⟨*tests*⟩ are checked (to be detailed later) and if one of the tests succeeds, the ⟨*code*⟩ is executed. If none of the tests succeeds, the ⟨*else code*⟩ is executed.

Example: `\pgfcalendarifdate{2007-02-07}{Wednesday}{Is a Wednesday}{Is not a Wednesday}` yields `Is a Wednesday`.

The ⟨*tests*⟩ is a comma-separated list of key–value pairs. The following are defined by default:

- `all` This test is passed by all dates.
- `Monday` This test is passed by all dates that are Mondays.
- `Tuesday` as above.
- `Wednesday` as above.
- `Thursday` as above.
- `Friday` as above.
- `Saturday` as above.
- `Sunday` as above.
- `workday` Passed by Mondays, Tuesdays, Wednesdays, Thursdays, and Fridays.
- `weekend` Passed by Saturdays and Sundays.
- `equals=`⟨*reference*⟩ The ⟨*reference*⟩ can be in one of two forms: Either, it is a full ISO format date like `2007-01-01` or the year may be missing as in `12-31`. In the first case, the test is passed if ⟨*date*⟩ is the same as ⟨*reference*⟩. In the second case, the test is passed if the month and day part of ⟨*date*⟩ is the same as ⟨*reference*⟩.

  For example, the test `equals=2007-01-10` will only be passed by this particular date. The test `equals=05-01` will be passed by every first of May on any year.

- `at least=`⟨*reference*⟩ This test works similarly to the `equals` test, only it is checked whether ⟨*date*⟩ is equal to ⟨*reference*⟩ or to any later date. Again, the ⟨*reference*⟩ can be a full date like `2007-01-01` or a short version like `07-01`. For example, `at least=07-01` is true for every day in the second half of any year.
- `at most=`⟨*reference*⟩ as above.
- `between=`⟨*start reference*⟩ `and` ⟨*end reference*⟩ This test checks whether the current date lies between the two given reference dates. Both full and short version may be given.

  For example `between=2007-01-01 and 2007-02-28` is true for the days in January and February of 2007.

  For another example, `between=05-01 and 05-07` is true for the days of the first week of May of any year.
- `day of month=`⟨*number*⟩ Passed by the day of month of the ⟨*date*⟩ that is ⟨*number*⟩. For example, the test `day of month=1` is passed by every first of every month.
- `end of month=`⟨*number*⟩ Passed by the day of month of the ⟨*date*⟩ that is ⟨*number*⟩ from the end of the month. For example, the test `end of month=1` is passed by the last day of every month, the test `end of month=2` is passed by the second last day of every month. If ⟨*number*⟩ is omitted, it is assumed to be `1`.
- `Easter=`⟨*number*⟩ This test checks whether the given date is Easter Sunday. The optional number can be used for offsets from Easter Sunday, e.g. `Easter=-3` for Maundy Thursday, `Easter=-2` for Good Friday, `Easter=1` for Easter Monday. Since the dates of other Christian holidays are determined by the date of Easter, these can be accessed as well, e.g. `Easter=39` for Feast of the Ascension, `Easter=49` for Pentecost, and `Easter=50` for Whit Monday.

In addition to the above checks, you can also define new checks. To do so, you must add a new key to the path `/pgf/calendar/` using the `\pgfkeys` command. The job of the code of this new key is to possibly set the TEX-if `\ifpgfcalendarmatches` to true (if it is already true, no action should be taken) to indicate that the ⟨*date*⟩ passes the test setup by this new key.

In order to perform the test, the key code needs to know the date that should be checked. The date is available through a macro, but a whole bunch of additional information about this date is also available through the following macros:

- `\pgfcalendarifdatejulian` is the Julian day number of the ⟨*date*⟩ to be checked.
- `\pgfcalendarifdateweekday` is the weekday of the ⟨*date*⟩ to be checked.
- `\pgfcalendarifdateyear` is the year of the ⟨*date*⟩ to be checked.
- `\pgfcalendarifdatemonth` is the month of the ⟨*date*⟩ to be checked.
- `\pgfcalendarifdateday` is the day of month of the ⟨*date*⟩ to be checked.

For example, let us define a new key that checks whether the ⟨*date*⟩ is a Workers day (May 1st). This can be done as follows:

```
\pgfkeys{/pgf/calendar/workers day/.code=%
{
  \ifnum\pgfcalendarifdatemonth=5\relax
    \ifnum\pgfcalendarifdateday=1\relax
      \pgfcalendarmatchestrue
    \fi
  \fi
}}
```

89.1.3 Typesetting Dates

`\pgfcalendarweekdayname{`⟨*week day number*⟩`}`

This command expands to a textual representation of the day of week, given by the ⟨*week day number*⟩. Thus, `\pgfcalendarweekdayname{0}` expands to `Monday` if the current language is English and to `Montag` if the current language is German, and so on. See Section 89.1.4 for more details on translations.

Example: `\pgfcalendarweekdayname{2}` yields `Wednesday`.

\pgfcalendarweekdayshortname{⟨*week day number*⟩}

This command works similarly to the previous command, only an abbreviated version of the week day is produced.

Example: `\pgfcalendarweekdayshortname{2}` yields `Wed`.

\pgfcalendarmonthname{⟨*month number*⟩}

This command expands to a textual representation of the month, which is given by the ⟨*month number*⟩.

Example: `\pgfcalendarmonthname{12}` yields `December`.

\pgfcalendarmonthshortname{⟨*month number*⟩}

As above, only an abbreviated version is produced.

Example: `\pgfcalendarmonthshortname{12}` yields `Dec`.

89.1.4 Localization

All textual representations of week days or months (like "Monday" or "February") are wrapped with `\translate` commands from the `translator` package (it this package is not loaded, no translation takes place). Furthermore, the `pgfcalendar` package will try to load the `translator-months-dictionary`, if the `translator` package is loaded.

If you want to use the `translator` package, it has to be loaded before the `pgfcalendar` package or, when you are using the `calendar` Ti*k*Z library, before `tikz`. Otherwise it will not be properly detected.

The net effect of all this is that all dates will be translated to the current language setup in the `translator` package. See the documentation of this package for more details.

89.2 Typesetting Calendars

\pgfcalendar{⟨*prefix*⟩}{⟨*start date*⟩}{⟨*end date*⟩}{⟨*rendering code*⟩}

This command can be used to typeset a calendar. It is a very general command, the actual work has to be done by giving clever implementations of ⟨*rendering code*⟩. Note that this macro need *not* be called inside a `{pgfpicture}` environment (even though it typically will be) and you can use it to typeset calendars in normal TEX or using packages other than PGF.

Basic typesetting process. A calendar is typeset as follows: The ⟨*start date*⟩ and ⟨*end date*⟩ specify a range of dates. For each date in this range the ⟨*rendering code*⟩ is executed with certain macros setup to yield information about the *current date* (the current date in the enumeration of dates of the range). Typically, the ⟨*rendering code*⟩ places nodes inside a picture, but it can do other things as well. Note that it is also the job of the ⟨*rendering code*⟩ to position the calendar correctly.

The different calls of the ⟨*rending code*⟩ are not surrounded by TEX groups (though you can do so yourself, of course). This means that settings can accumulate between different calls, which is often desirable and useful.

Information about the current date. Inside the ⟨*rendering code*⟩, different macros can be access:

- `\pgfcalendarprefix` The ⟨*prefix*⟩ parameter. This prefix is recommended for nodes inside the calendar, but you have to use it yourself explicitly.
- `\pgfcalendarbeginiso` The ⟨*start date*⟩ of range being typeset in ISO format (like 2006-01-10).
- `\pgfcalendarbeginjulian` Julian day number of ⟨*start date*⟩.
- `\pgfcalendarendiso` The ⟨*end date*⟩ of range being typeset in ISO format.
- `\pgfcalendarendjulian` Julian day number of ⟨*end date*⟩.
- `\pgfcalendarcurrentjulian` This TEX count holds the Julian day number of the day currently being rendered.
- `\pgfcalendarcurrentweekday` The weekday (a number with zero representing Monday) of the current date.
- `\pgfcalendarcurrentyear` The year of the current date.
- `\pgfcalendarcurrentmonth` The month of the current date (always two digits with a leading zero, if necessary).

- `\pgfcalendarcurrentday` The day of month of the current date (always two digits).

The `\ifdate` command. Inside the `\pgfcalendar` the macro `\ifdate` is available locally:

`\ifdate{`⟨*tests*⟩`}{`⟨*code*⟩`}{`⟨*else code*⟩`}`

This command has the same effect as calling `\pgfcalendarifdate` for the current date.

Examples. In a first example, let us create a very simple calendar: It just lists the dates in a certain range.

20 21 22 23 24 25 26 27 28 29 30 31 01 02 03 04 05 06 07 08 09 10

```
\usepackage {pgfcalendar}
\pgfcalendar{cal}{2007-01-20}{2007-02-10}{\pgfcalendarcurrentday\ }
```

Let us now make this a little more interesting: Let us add a line break after each Sunday.

20 21
22 23 24 25 26 27 28
29 30 31 01 02 03 04
05 06 07 08 09 10

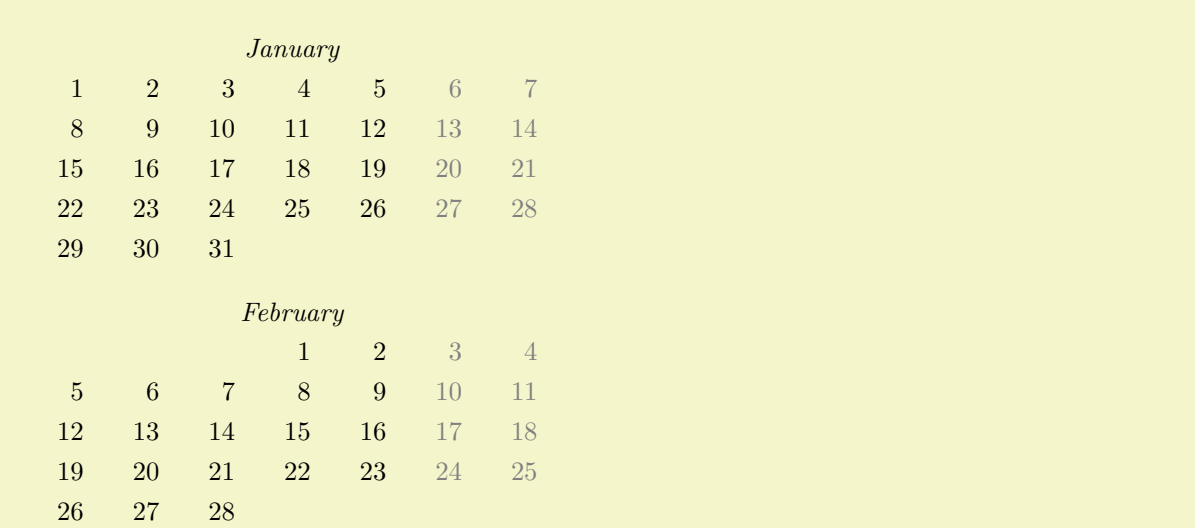
```
\usepackage {pgfcalendar}
\pgfcalendar{cal}{2007-01-20}{2007-02-10}
{
  \pgfcalendarcurrentday\
  \ifdate{Sunday}{\par}{}
}
```

We now want to have all Mondays to be aligned on a column. For this, different approaches work. Here is one based positioning each day horizontally using a skip.

20 21
22 23 24 25 26 27 28
29 30 31 01 02 03 04
05 06 07 08 09 10

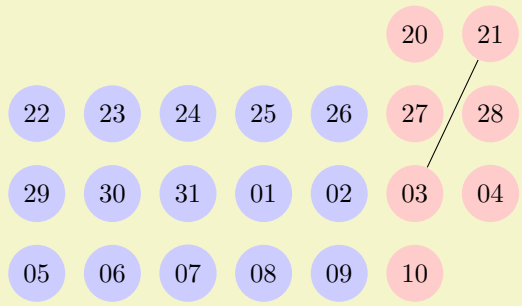
```
\usepackage {pgfcalendar}
\pgfcalendar{cal}{2007-01-20}{2007-02-10}
{%
  \leavevmode%
  \hbox to0pt{\hskip\pgfcalendarcurrentweekday cm\pgfcalendarcurrentday\hss}%
  \ifdate{Sunday}{\par}{}%
}
```

Let us now typeset two complete months.




```
\usepackage {pgfcalendar}
\pgfcalendar{cal}{2007-01-01}{2007-02-28}{%
  \ifdate{day of month=1}{
    \par\bigskip\hbox to7.5cm{\itshape\hss\pgfcalendarshorthand mt\hss}\par
  }{}%
  \leavevmode%
  {%
    \ifdate{weekend}{\color{black!50}}{\color{black}}%
    \hbox to0pt{%
      \hskip\pgfcalendarcurrentweekday cm%
      \hbox to1cm{\hss\pgfcalendarshorthand d-}\hss%
    }%
  }%
  \ifdate{Sunday}{\par}{}%
}
```

For our final example, we use a `{tikzpicture}`.




```
\usepackage {pgfcalendar}
\begin{tikzpicture}
  \pgfcalendar{cal}{2007-01-20}{2007-02-10}{%
    \ifdate{workday}
      {\tikzset{filling/.style={fill=blue!20}}}
      {\tikzset{filling/.style={fill=red!20}}}
    \node (\pgfcalendarsuggestedname) at (\pgfcalendarcurrentweekday,0)
      [anchor=base,circle,filling] {\pgfcalendarcurrentday};
    \ifdate{Sunday}{\pgftransformyshift{-3em}}{}%
  }
  \draw (cal-2007-01-21) -- (cal-2007-02-03);
\end{tikzpicture}
```

\pgfcalendarshorthand{⟨*kind*⟩}{⟨*representation*⟩}

This command can be used inside a `\pgfcalendar`, where it will expand to a representation of the current day, month, year or day of week, depending on whether ⟨*kind*⟩ is `d`, `m`, `y` or `w`. The ⟨*representation*⟩ can be one of the following: `-`, `=`, `0`, `.`, and `t`. They have the following meanings:

- The minus sign selects the shortest numerical representation possible (no leading zeros).
- The equal sign also selects the shortest numerical representation, but a space is added to single digit days and months (thereby ensuring that they have the same length as other days).
- The zero digit selects a two-digit numerical representation for days and months. For years it is allowed, but has no effect.
- The letter `t` selects a textual representation.
- The dot selects an abbreviated textual representation.

Normally, you should say `\let\%=\pgfcalendarshorthand` locally, so that you can write `\%wt` instead of the much more cumbersome `\pgfcalendarshorthand{w}{t}`.

ISO form: 2007-01-20, long form: Saturday, January 20, 2007

```
\usepackage {pgfcalendar}
\let\%=\pgfcalendarshorthand
\pgfcalendar{cal}{2007-01-20}{2007-01-20}
{ ISO form: \%y0-\%m0-\%d0, long form: \%wt, \%mt \%d-, \%y0}
```

\pgfcalendarsuggestedname

This macro expands to a suggested name for nodes representing days in a calendar. If the ⟨*prefix*⟩ is empty, it expands to the empty string, otherwise it expands to the ⟨*prefix*⟩ of the calendar, followed by a hyphen, followed by the ISO format version of the date. Thus, when the date `2007-01-01` is typeset in a calendar for the prefix `mycal`, the macro expands to `mycal-2007-01-01`.

90 Page Management

This section describes the pgfpages package. Although this package is not concerned with creating pictures, its implementation relies so heavily on PGF that it is documented here. Currently, pgfpages only works with LaTeX, but if you are adventurous, feel free to hack the code so that it also works with plain TeX.

The aim of pgfpages is to provide a flexible way of putting multiple pages on a single page *inside TeX*. Thus, pgfpages is quite different from useful tools like psnup or pdfnup insofar as it creates its output in a single pass. Furthermore, it works uniformly with both latex and pdflatex, making it easy to put multiple pages on a single page without any fuss.

A word of warning: *using pgfpages will destroy hyperlinks.* Actually, the hyperlinks are not destroyed, only they will appear at totally wrong positions on the final output. This is due to a fundamental flaw in the PDF specification: In PDF the bounding rectangle of a hyperlink is given in "absolute page coordinates" and translations or rotations do not affect them. Thus, the transformations applied by pgfpages to put the pages where you want them are (cannot, even) be applied to the coordinates of hyperlinks. It is unlikely that this will change in the foreseeable future.

90.1 Basic Usage

The internals of pgfpages are complex since the package can do all sorts of interesting tricks. For this reason, so-called *layouts* are predefined that set up all option in appropriate ways.

You use a layout as follows:

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The layout 2 on 1 puts two pages on a single page. The option a4paper tells pgfpages that the *resulting* page (called the *physical* page in the following) should be a4paper and it should be landscape (which is quite logical since putting two portrait pages next to each other gives a landscape page). Normally, the *logical* pages, that is, the pages that TeX "thinks" that it is typesetting, will have the same sizes, but this need not be the case. pgfpages will automatically scale down the logical pages such that two logical pages fit next to each other inside a DIN A4 page.

The border shrink tells pgfpages that it should add an additional 5mm to the shrinking such that a 5mm-wide border is shown around the resulting logical pages.

As a second example, let us put two pages produced by the BEAMER class on a single page:

```
\documentclass{beamer}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,border shrink=5mm]

\begin{document}
\begin{frame}
  This text is shown at the top.
\end{frame}
\begin{frame}
  This text is shown at the bottom.
\end{frame}
\end{document}
```

Note that we do not use the landscape option since BEAMER's logical pages are already in landscape mode and putting two landscape pages on top of each other results in a portrait page. However, if you had used the 4 on 1 layout, you would have had to add landscape once more, using the 6 on 1 or 8 on 1 you must not, using 16 on 1 you need it yet again. And, no, there is no 32 on 1 layout.

Another word of caution: *using pgfpages will produce wrong page numbers in the .aux file.* The reason is that TeX instantiates the page numbers when writing an .aux file only when the physical page is shipped out. Fortunately, this problem is easy to fix: First, typeset our file normally without using the \pgfpagesuselayout command (just put the comment marker % before it) Then, rerun TeX with the

\pgfpagesuselayout command included and add the command \nofiles. This command ensures that the .aux file is not modified, which is exactly what you want. So, to typeset the above example, you should actually first TEX the following file:

```
\documentclass{article}

\usepackage{pgfpages}
%%\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
%%\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

and then typeset

```
\documentclass{article}

\usepackage{pgfpages}
\pgfpagesuselayout{2 on 1}[a4paper,landscape,border shrink=5mm]
\nofiles

\begin{document}
This text is shown on the left.
\clearpage
This text is shown on the right.
\end{document}
```

The final basic example is the `resize to` layout (it works a bit like a hypothetical `1 on 1` layout). This layout resizes the logical page such that is fits the specified physical size. Since this does not change the page numbering, you need not worry about the `.aux` files with this layout. For example, adding the following lines will ensure that the physical output will fit on DIN A4 paper:

```
\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[a4paper]
```

This can be very useful when you have to handle lots of papers that are typeset for, say, letter paper and you have an A4 printer or the other way round. For example, the following article will be fit for printing on letter paper:

```
\documentclass[a4paper]{article}
%% a4 is currently the logical size and also the physical size

\usepackage{pgfpages}
\pgfpagesuselayout{resize to}[letterpaper]
%% a4 is still the logical size, but letter is the physical one

\begin{document}
  \title{My Great Article}
...
\end{document}
```

90.2 The Predefined Layouts

This section explains the predefined layouts in more detail. You select a layout using the following command:

\pgfpagesuselayout{⟨*layout*⟩}[⟨*options*⟩]

Installs the specified ⟨*layout*⟩ with the given ⟨*options*⟩. The predefined layouts and their permissible options are explained below.

If this function is called multiple times, only the last call "wins". You can thereby overwrite any previous settings. In particular, layouts *do not* accumulate.

Example: `\pgfpagesuselayout{resize to}[a4paper]`

\pgfpagesuselayout{resize to}[⟨*options*⟩]

This layout is used to resize every logical page to a specified physical size. To determine the target size, the following options may be given:

- `physical paper height=`⟨*size*⟩ sets the height of the physical page size to ⟨*size*⟩.
- `physical paper width=`⟨*size*⟩ sets the width of the physical Pappe size to ⟨*size*⟩.
- `a0paper` sets the physical page size to DIN A0 paper.
- `a1paper` sets the physical page size to DIN A1 paper.
- `a2paper` sets the physical page size to DIN A2 paper.
- `a3paper` sets the physical page size to DIN A3 paper.
- `a4paper` sets the physical page size to DIN A4 paper.
- `a5paper` sets the physical page size to DIN A5 paper.
- `a6paper` sets the physical page size to DIN A6 paper.
- `letterpaper` sets the physical page size to the American letter paper size.
- `legalpaper` sets the physical page size to the American legal paper size.
- `executivepaper` sets the physical page size to the American executive paper size.
- `landscape` swaps the height and the width of the physical paper.
- `border shrink=`⟨*size*⟩ additionally reduces the size of the logical page on the physical page by ⟨*size*⟩.

`\pgfpagesuselayout{2 on 1}[`⟨*options*⟩`]`

Puts two logical pages alongside each other on each physical page if the logical height is larger than the logical width (logical pages are in portrait mode). Otherwise, two logical pages are put on top of each other (logical pages are in landscape mode). When using this layout, it is advisable to use the `\nofiles` command, but this is not done automatically.

The same ⟨*options*⟩ as for the `resize to` layout can be used, plus the following option:

- `odd numbered pages right` places the first page on the right.

`\pgfpagesuselayout{4 on 1}[`⟨*options*⟩`]`

Puts four logical pages on a single physical page. The same ⟨*options*⟩ as for the `resize to` layout can be used.

`\pgfpagesuselayout{6 on 1}[`⟨*options*⟩`]`

Puts six logical pages on a single physical page.

`\pgfpagesuselayout{8 on 1}[`⟨*options*⟩`]`

Puts eight logical pages on a single physical page. As for `2 on 1` and `4 on 1`, the orientation depends on whether the logical pages are in landscape mode or in portrait mode.

`\pgfpagesuselayout{16 on 1}[`⟨*options*⟩`]`

This is for the CEO.

`\pgfpagesuselayout{rounded corners}[`⟨*options*⟩`]`

This layout adds "rounded corners" to every page, which, supposedly, looks nicer during presentations with projectors (personally, I doubt this). This is done by (possibly) resizing the page to the physical page size. Then four black rectangles are drawn in each corner. Next, a clipping region is set up that contains all of the logical page except for little rounded corners. Finally, the logical page is drawn and clipped against the clipping region.

Note that every logical page should fill its background for this to work.

In addition to the ⟨*options*⟩ that can be given to `resize to`, the following options may be given.

- `corner width=`⟨*size*⟩ specifies the size of the corner.

```
\documentclass{beamer}
\usepackage{pgfpages}
\pgfpagesuselayout{rounded corners}[corner width=5pt]
\begin{document}
...
\end{document}
```

`\pgfpagesuselayout{two screens with lagging second}[⟨options⟩]`

This layout puts two logical pages alongside each other. The second page always shows what the main page showed on the previous physical page. Thus, the second page "lags behind" the main page. This can be useful when you have two projectors attached to your computer and can show different parts of a physical page on different projectors.

The following ⟨*options*⟩ may be given:

- `second right` puts the second page right of the main page. This will make the physical pages twice as wide as the logical pages, but it will retain the height.
- `second left` puts the second page left, otherwise it behaves the same as `second right`.
- `second bottom` puts the second page below the main page. This make the physical pages twice as high as the logical ones.
- `second top` works like `second bottom`.

`\pgfpagesuselayout{two screens with optional second}[⟨options⟩]`

This layout works similarly to `two screens with lagging second`. The difference is that the contents of the second screen only changes when one of the commands `\pgfshipoutlogicalpage{2}{⟨box⟩}` or `\pgfcurrentpagewillbelogicalpage{2}` is called. The first puts the given ⟨*box*⟩ on the second page. The second specifies that the current page should be put there, once it is finished.

The same options as for `two screens with lagging second` may be given.

You can define your own predefined layouts using the following command:

`\pgfpagesdeclarelayout{⟨layout⟩}{⟨before actions⟩}{⟨after actions⟩}`

This command predefines a ⟨*layout*⟩ that can later be installed using the `\pgfpagesuselayout` command.

When `\pgfpagesuselayout{⟨layout⟩}[⟨options⟩]` is called, the following happens: First, the ⟨*before actions*⟩ are executed. They can be used, for example, to set up default values for keys. Next, `\setkeys{pgfpagesuselayoutoption}{⟨options⟩}` is executed. Finally, the ⟨*after actions*⟩ are executed.

Here is an example:

```
\pgfpagesdeclarelayout{resize to}
{
  \def\pgfpageoptionborder{0pt}
}
{
  \pgfpagesphysicalpageoptions
  {%
    logical pages=1,%
    physical height=\pgfpageoptionheight,%
    physical width=\pgfpageoptionwidth%
  }
  \pgfpageslogicalpageoptions{1}
  {%
    resized width=\pgfphysicalwidth,%
    resized height=\pgfphysicalheight,%
    border shrink=\pgfpageoptionborder,%
    center=\pgfpoint{.5\pgfphysicalwidth}{.5\pgfphysicalheight}%
  }%
}
```

90.3 Defining a Layout

If none of the predefined layouts meets your problem or if you wish to modify them, you can create layouts from scratch. This section explains how this is done.

Basically, `pgfpages` hooks into TEX's `\shipout` function. This function is called whenever TEX has completed typesetting a page and wishes to send this page to the `.dvi` or `.pdf` file. The `pgfpages` package redefines this command. Instead of sending the page to the output file, `pgfpages` stores it in an internal box and then acts as if the page had been output. When TEX tries to output the next page using `\shipout`, this call is once more intercepted and the page is stored in another box. These boxes are called *logical pages*.

At some point, enough logical pages have been accumulated such that a *physical page* can be output. When this happens, `pgfpages` possibly scales, rotates, and translates the logical pages (and possibly even does further modifications) and then puts them at certain positions of the *physical* page. Once this page is fully assembled, the "real" or "original" `\shipout` is called to send the physical page to the output file.

In reality, things are slightly more complicated. First, once a physical page has been shipped out, the logical pages are usually voided, but this need not be the case. Instead, it is possible that certain logical pages just retain their contents after the physical page has been shipped out and these pages need not be filled once more before a physical shipout can occur. However, the contents of these logical pages can still be changed using special commands. It is also possible that after a shipout certain logical pages are filled with the contents of *other* logical pages.

A *layout* defines for each logical page where it will go on the physical page and which further modifications should be done. The following two commands are used to define the layout:

`\pgfpagesphysicalpageoptions{`⟨*options*⟩`}`

This command sets the characteristics of the "physical" page. For example, it is used to specify how many logical pages there are and how many logical pages must be accumulated before a physical page is shipped out. How each individual logical page is typeset is specified using the command `\pgfpageslogicalpageoptions`, described later.

Example: A layout for putting two portrait pages on a single landscape page:

```
\pgfpagesphysicalpageoptions
{%
  logical pages=2,%
  physical height=\paperwidth,%
  physical width=\paperheight,%
}

\pgfpageslogicalpageoptions{1}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
\pgfpageslogicalpageoptions{2}
{%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
  center=\pgfpoint{.75\pgfphysicalwidth}{.5\pgfphysicalheight}%
}%
```

The following ⟨*options*⟩ may be set:

- `logical pages=`⟨*logical pages*⟩ specified how many logical pages there are, in total. These are numbered 1 to ⟨*logical pages*⟩.
- `first logical shipout=`⟨*first*⟩. See the next option. By default, ⟨*first*⟩ is 1.
- `last logical shipout=`⟨*last*⟩. Together with the previous option, these two options define an interval of pages inside the range 1 to ⟨*logical pages*⟩. Only this range is used to store the pages that are shipped out by TEX. This means that after a physical shipout has just occurred (or at the beginning), the first time TEX wishes to perform a shipout, the page to be shipped out is stored in logical page ⟨*first*⟩. The next time TEX performs a shipout, the page is stored in logical page ⟨*first*⟩ + 1 and so on, until the logical page ⟨*last*⟩ is also filled. Once this happens, a physical shipout occurs and the process starts once more.

  Note that logical pages that lie outside the interval between ⟨*first*⟩ and ⟨*last*⟩ are filled only indirectly or when special commands are used.

  By default, ⟨*last*⟩ equals ⟨*logical pages*⟩.
- `current logical shipout=`⟨*current*⟩ changes an internal counter such that TEX's next logical shipout will be stored in logical page ⟨*current*⟩.

  This option can be used to "warp" the logical page filling mechanism to a certain page. You can both skip logical pages and overwrite already filled logical pages. After the logical page ⟨*current*⟩ has been filled, the internal counter is incremented normally as if the logical page ⟨*current*⟩ had been "reached" normally. If you specify a ⟨*current*⟩ larger than ⟨*last*⟩, a physical shipout will occur after the logical page ⟨*current*⟩ has been filled.

- `physical height=`⟨*height*⟩ specifies the height of the physical pages. This height is typically different from the normal `\paperheight`, which is used by TEX for its typesetting and page breaking purposes.
- `physical width=`⟨*width*⟩ specifies the physical width.

`\pgfpageslogicalpageoptions{`⟨*logical page number*⟩`}{`⟨*options*⟩`}`

This command is used to specify where the logical page number ⟨*logical page number*⟩ will be placed on the physical page. In addition, this command can be used to install additional "code" to be executed when this page is put on the physical page.

The number ⟨*logical page number*⟩ should be between 1 and ⟨*logical pages*⟩, which has previously been installed using the `\pgfpagesphysicalpageoptions` command.

The following ⟨*options*⟩ may be given:

- `center=`⟨*pgf point*⟩ specifies the center of the logical page inside the physical page as a PGF-point. The origin of the coordinate system of the physical page is at the *lower* left corner.

```
\pgfpageslogicalpageoptions{1}
{% center logical page on middle of left side
  center=\pgfpoint{.25\pgfphysicalwidth}{.5\pgfphysicalheight}%
  resized width=.5\pgfphysicalwidth,%
  resized height=\pgfphysicalheight,%
}
```

- `resized width=`⟨*size*⟩ specifies the width that the logical page should have *at most* on the physical page. To achieve this width, the pages is scaled down appropriately *or more*. The "or more" part can happen if the `resize height` option is also used. In this case, the scaling is chosen such that both the specified height and width are met. The aspect ratio of a logical page is not modified.
- `resized height=`⟨*height*⟩ specifies the maximum height of the logical page.
- `original width=`⟨*width*⟩ specifies the width the TEX "thinks" that the logical page has. This width is `\paperwidth` at the point of invocation, by default. Note that setting this width to something different from `\paperwidth` does *not* change the `\pagewidth` during TEX's typesetting. You have to do that yourself.

  You need this option only for special logical pages that have a height or width different from the normal one and for which you will (later on) set these sizes yourself.
- `original height=`⟨*height*⟩ works like `original width`.
- `scale=`⟨*factor*⟩ scales the page by at least the given ⟨*factor*⟩. A ⟨*factor*⟩ of `0.5` will half the size of the page, a factor or `2` will double the size. "At least" means that if options like `resize height` are given and if the scaling required to meet that option is less than ⟨*factor*⟩, that other scaling is used instead.
- `xscale=`⟨*factor*⟩ scales the logical page along the x-axis by the given ⟨*factor*⟩. This scaling is done independently of any other scaling. Mostly, this option is useful for a factor of `-1`, which flips the page along the y-axis. The aspect ratio is not kept.
- `yscale=`⟨*factor*⟩ works like `xscale`, only for the y-axis.
- `rotation=`⟨*degree*⟩ rotates the page by ⟨*degree*⟩ around its center. Use a degree of `90` or `-90` to go from portrait to landscape and back. The rotation need not be a multiple of `90`.
- `copy from=`⟨*logical page number*⟩. Normally, after a physical shipout has occurred, all logical pages are voided in a loop. However, if this option is given, the current logical page is filled with the contents of the old logical page number ⟨*logical page number*⟩.

  Example: Have logical page 2 retain its contents:

```
\pgfpageslogicalpageoptions{2}{copy from=2}
```

  Example: Let logical page 2 show what logical page 1 showed on the just-shipped-out physical page:

```
\pgfpageslogicalpageoptions{2}{copy from=1}
```

- `border shrink=`⟨*size*⟩ specifies an additional reduction of the size to which the page is page is scaled.
- `border code=`⟨*code*⟩. When this option is given, the ⟨*code*⟩ is executed before the page box is inserted with a path preinstalled that is a rectangle around the current logical page. Thus, setting ⟨*code*⟩ to `\pgfusepath{stroke}` draws a rectangle around the logical page. Setting ⟨*code*⟩ to `\pgfsetlinewidth{3pt}\pgfusepath{stroke}` results in a thick (ugly) frame. Adding dashes and filling can result in arbitrarily funky and distracting borders.

  You can also call `\pgfusepath{discard}` and add your own path construction code (for example to paint a rectangle with rounded corners). The coordinate system is set up in such a way that a rectangle starting at the origin and having the height and width of TEX-box 0 will result in a rectangle filling exactly the logical page currently being put on the physical page. The logical page is inserted *after* these commands have been executed.

  Example: Add a rectangle around the page:

  ```
  \pgfpageslogicalpageoptions{1}{border code=\pgfusepath{stroke}}
  ```

- `corner width=`⟨*size*⟩ adds black "rounded corners" to the page. See the description of the predefined layout `rounded corners` on page 1018.

90.4 Creating Logical Pages

Logical pages are created whenever TEX thinks that a page is full and performs a `\shipout` command. This will cause `pgfpages` to store the box that was supposed to be shipped out internally until enough logical pages have been collected such that a physical shipout can occur.

Normally, whenever a logical shipout occurs, that current page is stored in logical page number ⟨*current logical page*⟩. This counter is then incremented, until it is larger than ⟨*last logical shipout*⟩. You can, however, directly change the value of ⟨*current logical page*⟩ by calling `\pgfpagesphysicalpageoptions`.

Another way to set the contents of a logical page is to use the following command:

`\pgfpagesshipoutlogicalpage{`⟨*number*⟩`}`⟨*box*⟩

This command sets to logical page ⟨*number*⟩ to ⟨*box*⟩. The ⟨*box*⟩ should be the code of a TEX box command. This command does not influence the counter ⟨*current logical page*⟩ and does not cause a physical shipout.

```
\pgfpagesshipoutlogicalpage{0}\vbox{Hi!}
```

This command can be used to set the contents of logical pages that are normally not filled.

The final way of setting a logical page is using the following command:

`\pgfpagescurrentpagewillbelogicalpage{`⟨*number*⟩`}`

When the current TEX page has been typeset, it will be become the given logical page ⟨*number*⟩. This command "interrupts" the normal order of logical pages, that is, it behaves like the previous command and does not update the ⟨*current logical page*⟩ counter.

```
\pgfpagesuselayout{two screens with optional second}
...
Text for main page.
\clearpage

\pgfpagescurrentpagewillbelogicalpage{2}
Text that goes to second page
\clearpage

Text for main page.
```

91 Extended Color Support

This section documents the package `xxcolor`, which is currently distributed as part of PGF. This package extends the `xcolor` package, written by Uwe Kern, which in turn extends the `color` package. I hope that the commands in `xxcolor` will some day migrate to `xcolor`, such that this package becomes superfluous.

The main aim of the `xxcolor` package is to provide an environment inside which all colors are "washed out" or "dimmed". This is useful in numerous situations and must typically be achieved in a roundabout manner if such an environment is not available.

`\begin{colormixin}{`⟨*mix-in specification*⟩`}`
⟨*environment contents*⟩
`\end{colormixin}`

The ⟨*mix-in specification*⟩ is applied to all colors inside the environment. At the beginning of the environment, the mix-in is applied to the current color, i.e., the color that was in effect before the environment started. A ⟨*mix-in specification*⟩ is a number between 0 and 100 followed by an exclamation mark and a color name. When a `\color` command is encountered inside a mix-in environment, the number states what percentage of the desired color should be used. The rest is "filled up" with the color given in the mix-in specification. Thus, a mix-in specification like `90!blue` will mix in 10% of blue into everything, whereas `25!white` will make everything nearly white.

Red text, washed-out red text, washed-out blue text, dark washed-out blue text, dark washed-out green text, back to washed-out blue text, and back to red.

```
\usepackage {xxcolor}
\begin{minipage}{3.5cm}\raggedright
\color{red}Red text,
\begin{colormixin}{25!white}
  washed-out red text,
  \color{blue}washed-out blue text,
  \begin{colormixin}{25!black}
    dark washed-out blue text,
    \color{green}dark washed-out green text,
  \end{colormixin}%
  back to washed-out blue text,
\end{colormixin}%
and back to red.
\end{minipage}%
```

Note that the environment only changes colors that have been installed using the standard LaTeX `\color` command. In particular, the colors in images are not changed. There is, however, some support offered by the commands `\pgfuseimage` and `\pgfuseshading`. If the first command is invoked inside a `colormixin` environment with the parameter, say, `50!black` on an image with the name `foo`, the command will first check whether there is also a defined image with the name `foo.!50!black`. If so, this image is used instead. This allows you to provide a different image for this case. If you nest `colormixin` environments, the different mix-ins are all appended. For example, inside the inner environment of the above example, `\pgfuseimage{foo}` would first check whether there exists an image named `foo.!25!white!25!black`.

`\colorcurrentmixin`

Expands to the current accumulated mix-in. Each nesting of a `colormixin` adds a mix-in to this list.

!75!white should be "!75!white"
!75!black!75!white should be "!75!black!75!white"
!50!white!75!black!75!white should be "!50!white!75!black!75!white"

```
\usepackage {xxcolor} \usepackage {calc}
\begin{minipage}{\linewidth-6pt}\raggedright
\begin{colormixin}{75!white}
  \colorcurrentmixin\ should be ``!75!white''\par
  \begin{colormixin}{75!black}
    \colorcurrentmixin\ should be ``!75!black!75!white''\par
    \begin{colormixin}{50!white}
      \colorcurrentmixin\ should be ``!50!white!75!black!75!white''\par
    \end{colormixin}
  \end{colormixin}
\end{colormixin}
\end{minipage}
```

92 Parser Module

```
\usepgfmodule{parser}  % LaTeX and plain TeX when using pure PGF
\usepgfmodule[parser]  % ConTeXt when using pure PGF
```

This module defines some commands for creating a simple letter-by-letter parser.

```
\usepackage{pgfparser} % LaTeX
\input{pgfparser.tex} % plain TeX
\usemodule[pgfparser]  % ConTeXt
```

Because the parser module is almost independent of the rest of PGF, it can also be used as a standalone package with minimal dependencies.

This module provides commands for defining a parser that scans some given text letter-by-letter. For each letter, some code is executed and, possibly a state-switch occurs. The code for each letter might take mandatory or optional arguments. The parsing process ends when a final state has been reached, and optionally some code is executed afterwards. Each newly defined parser by default ignores space tokens, if you want to change that you'll have to explicitly define an action for blank spaces (with `\pgfparserdef`).

`\pgfparserparse{`⟨*parser name*⟩`}`⟨*text*⟩

This command is used to parse the ⟨*text*⟩ using the (previously defined) parser named ⟨*parser name*⟩.

The ⟨*text*⟩ is not contained in curly braces, rather it is all the text that follows. The end of the text is determined implicitly, namely when the final state of the parser has been reached. If you defined a final action for the parser using `\pgfparserdeffinal` it is executed now.

The parser works as follows: At any moment, it is in a certain *state*, initially this state is called `initial`. Then, the first letter of the ⟨*text*⟩ is examined (using the `\futurelet` command). For each possible state and each possible letter, some action code is stored in the parser in a table. This code is then executed. This code may, but need not, trigger a *state switch*, causing a new state to be set. The parser then moves on to the next character of the text and repeats the whole procedure, unless it is in the state `final`, which causes the parsing process to stop immediately.

In the following example, the parser counts the number of `a`'s in the text, ignoring any `b`'s. The ⟨*text*⟩ ends with the first `c`.

cccThere are 9 a's.

```
\usepgfmodule {parser}
\newcount\mycount
\pgfparserdef{myparser}{initial}{the letter a}%
{\advance\mycount by 1\relax}%
\pgfparserdef{myparser}{initial}{the letter b}%
{} % do nothing
\pgfparserdef{myparser}{initial}{the letter c}%
{\pgfparserswitch{final}}% done!

\pgfparserparse{myparser}aabaababababbbbbabaabcccc
There are \the\mycount\ a's.
```

`\pgfparserdef{`⟨*parser name*⟩`}{`⟨*state*⟩`}`⟨*symbol meaning*⟩`[`⟨*arguments*⟩`]{`⟨*action*⟩`}`

This command should be used repeatedly to define a parser named ⟨*parser name*⟩. With a call to this command you specify that the ⟨*parser name*⟩ should do the following: When it is in state ⟨*state*⟩ and reads the letter ⟨*symbol meaning*⟩, perform the code stored in ⟨*action*⟩.

The ⟨*symbol meaning*⟩ must be the text that results from applying the TeX command `\meaning` to the given character. For instance, `\meaning a` yields `the letter a`, while `\meaning 1` yields `the character 1`. A space yields `blank space  `. Alternatively you can give the symbol you want without surrounding it in braces. So both `\pgfparserdef{myparser}{initial}{the letter a}{foo}` and `\pgfparserdef{myparser}{initial}a{foo}` define an ⟨*action*⟩ for `the letter a`. This short form works for most tokens, but not for a space (in which case you can use `\pgfparserdef{myparser}{initial}{blank space}{foo}`), and opening braces (in which case you can use `\pgfparserdef{myparser}{initial}{\meaning\bgroup}{foo}`, and one might prefer to use `\pgfparserdef{myparser}{initial}{\meaning\egroup}{foo}` for closing braces as well). You can as well define an action for a macro's meaning (note that macros with different names can have the same meaning), so things like `\pgfparserdef{myparser}{initial}\texttt{foo}` are possible as well.

The ⟨*action*⟩ might require arguments which you can specify in the optional ⟨*arguments*⟩ string. The argument string can contain up to nine argument specifications of the following types:

`m` a normal mandatory argument

`r`⟨*delim*⟩
: a mandatory argument which is read up to the ⟨*delim*⟩

`o` an optional argument in `[]` defaulting to a special mark

`O{`⟨*default*⟩`}`
: like `o` but defaulting to ⟨*default*⟩

`d`⟨*delim1*⟩⟨*delim2*⟩
: an optional argument in ⟨*delim1*⟩ and ⟨*delim2*⟩ defaulting to a special mark

`D`⟨*delim1*⟩⟨*delim2*⟩`{`⟨*default*⟩`}`
: like `d` but defaulting to ⟨*default*⟩

`t`⟨*token*⟩
: tests whether the next letter is ⟨*token*⟩, if so gobbles it and the argument is set to a special mark.

So if you want to define an ⟨*action*⟩ that takes two mandatory arguments you use `[mm]`, if it should take an optional star, one optional argument in brackets that returns a marker if it's not used, one mandatory and finally an optional argument in parentheses that defaults to `something` you use `[t*omD(){something}]` as the argument string. If the argument should be anything up to a semicolon, you use `[r;]`. Spaces before argument specifications in the string are ignored. So `[r m]` will be one argument and read anything up to an `m`. Also spaces before any argument in the parsed letters are ignored, so if `a` was setup to take an optional argument the argument would be detected in `a []`. Like with normal LaTeX 2_ε optional arguments you have to protect nested brackets: `[a[bc]d]` would be read as `a[bc` with a trailing `d]`, *not* as `a[bc]d`. You'd have to use `[{a[bc]d}]` to get it correct.

Inside the ⟨*action*⟩ you can perform almost any kind of code. This code will not be surrounded by a scope, so its effect persists after the parsing is done. However, each time after the ⟨*action*⟩ is executed, control goes back to the parser. You should not launch a parser inside the ⟨*action*⟩ code, unless you put it in a scope.

When you use `all` as the ⟨*state*⟩, the ⟨*action*⟩ is performed in all states as a fallback, whenever ⟨*symbol meaning*⟩ is encountered. This means that when you do not specify anything explicitly for a state and a letter, but you do specify something for `all` and this letter, then the specified ⟨*action*⟩ will be used.

When the parser encounters a letter for which nothing is specified in the current state (neither directly nor indirectly via `all`), an error occurs. Additionally you can specify an action that is executed after the error is thrown using `\pgfparserdefunknown`. To suppress these errors (but not the action specified with `\pgfparserdefunknown`) you can use the `/pgfparser/silent` key or the `silent` key of the current ⟨*parser name*⟩.

`\pgfparserlet{`⟨*parser name 1*⟩`}{`⟨*state 1*⟩`}`⟨*symbol meaning 1*⟩`[`⟨*opt 1*⟩`][`⟨*opt 2*⟩`]`⟨*symbol meaning 2*⟩

If none of the optional arguments are given in the following explanation ⟨*parser name 2*⟩ and ⟨*state 2*⟩ are the same as ⟨*parser name 1*⟩ and ⟨*state 1*⟩. If only the first is given ⟨*state 2*⟩ equals ⟨*opt 1*⟩. If both are given ⟨*parser name 2*⟩ equals ⟨*opt 1*⟩ and ⟨*state 2*⟩ equals ⟨*opt 2*⟩.

Defines an action for ⟨*parser name 1*⟩ in ⟨*state 1*⟩ for the ⟨*symbol meaning 1*⟩ to do the same as the action of ⟨*parser name 2*⟩ in ⟨*state 2*⟩ for the ⟨*symbol meaning 2*⟩. For ⟨*symbol meaning 1*⟩ and ⟨*symbol meaning 2*⟩ the same parsing rules apply as for ⟨*symbol meaning*⟩ in `\pgfparserdef` so you either give the meaning in braces or just the symbol.

`\pgfparserdefunknown{`⟨*parser name*⟩`}{`⟨*state*⟩`}{`⟨*action*⟩`}`

With this macro you can define an ⟨*action*⟩ for the ⟨*parser name*⟩ parser in ⟨*state*⟩ if no action was defined for the letter which was encountered.

`\pgfparserdeffinal{`⟨*parser name*⟩`}{`⟨*action*⟩`}`

Every parser can call a final ⟨*action*⟩ after the state was switched to `final`. This ⟨*action*⟩ is executed after everything else, so you can use something that grabs more arguments if you want to.

`\pgfparserswitch{`⟨*state*⟩`}`

This command can be called inside the action code of a parser to cause a state switch to ⟨*state*⟩.

\pgfparserifmark{⟨*arg*⟩}{⟨*true*⟩}{⟨*false*⟩}

Remember that some of the optional argument types set special marks? With \pgfparserifmark you can test whether ⟨*arg*⟩ is such a mark. So if there was no optional argument for the argument types o and d the ⟨*true*⟩ branch will be executed, else the ⟨*false*⟩ branch. For the t type argument the ⟨*true*⟩ branch is executed if the token was encountered.

\pgfparserreinsert

You can use this as the final macro in an action of \pgfparserdef or \pgfparserdefunknown. This has the effect that the contents of \pgfparserletter will be parsed next. Without any redefinition the result will be that the last token will be parsed again. You can change the definition of \pgfparserletter just before \pgfparserreinsert as well to parse some specific tokens next.

\pgfparserstate

Expands to the current state of the parser.

\pgfparsertoken

This is the macro which is let to the following token with \futurelet. You can use it inside an action code.

\pgfparserletter

This macro stores the letter to which \pgfparsertoken was let. So if you'd use \pgfparserparse{foo}a this macro would be defined with \def\pgfparserletter{a}. This definition is done before any action code is executed. There are four special cases: If the next token is of category code 1, 2, 6, or 10, so with standard category codes the tokens {, }, #, and ␣ (a space), it would be treated differently. In those cases this macro expands to \bgroup, \egroup, ##, and ␣ for the categories 1, 2, 6, and 10, respectively.

\pgfparserset{⟨*key list*⟩}

The pgfparser module has a few keys you can access through this macro. It is just a shortcut for \pgfset{/pgfparser/.cd,#1}. The available keys are listed in subsection 92.1.

92.1 Keys of the Parser Module

/pgfparser/silent=⟨*boolean*⟩ (no default, initially false)

If true then no error will be thrown when a letter is parsed for which no action is specified, silently ignoring it. This holds true for every parser.

/pgfparser/status=⟨*boolean*⟩ (no default, initially false)

If true every parser prints a status message for every action executed. This might help in debugging and understanding what the parser does.

Additionally to those keys for every ⟨*parser name*⟩ for which \pgfparserdef, \pgfparserdefunknown or \pgfparserlet was run at least once the following will be defined:

/pgfparser/⟨*parser name*⟩/silent=⟨*boolean*⟩ (no default, initially false)

If true the parser ⟨*parser name*⟩ will silently ignore undefined letters. This is an individual equivalent of /pgfparser/silent for each defined parser.

92.2 Examples

The following example counts the different letters appearing in a more or less random string of letters. Every letter is counted only once, this is achieved by defining a new action for every encountered unknown letter that does nothing. We can define such rule without knowing which letter is used, because \pgfparsertoken has the same meaning as that letter.

13 different letters found

```
\usepgfmodule {parser}
\mycount=0
% using the shortcut syntax of just placing ; after the state
\pgfparserdef{different letters}{all};{\pgfparserswitch{final}}%
\pgfparserdefunknown{different letters}{all}%
  {\pgfparserdef{different letters}{all}\pgfparsertoken{}\advance\mycount1}%
\pgfparserdeffinal{different letters}%
  {\the\mycount\ different letters found}%
% don't throw errors for unknown letters
\pgfparserset{different letters/silent=true}%

\pgfparserparse{different letters}udiaternxqlchudiea;
```

Next we want to try something that uses some of the different argument types available.

nobody **will** *use* Parser

```
\usepgfmodule {parser}
% using the long syntax of \pgfparserdef
\pgfparserdef{arguments}{initial}{the letter a}[d()]
  {\pgfparserifmark{#1}{\textcolor{red}{\textit{use}}}{\textbf{#1}} }%
% using the shortcut syntax
\pgfparserdef{arguments}{initial}t[m]{\texttt{#1} }%
\pgfparserdef{arguments}{initial}c[t*0{blue}m]
  {\pgfparserifmark{#1}{#3}{\textcolor{#2}{#3}}}%
\pgfparserdef{arguments}{all};{\pgfparserswitch{final}}%

\pgfparserparse{arguments}t{nobody}a(will)ac[green]{P}c*{arse}c{r};
```

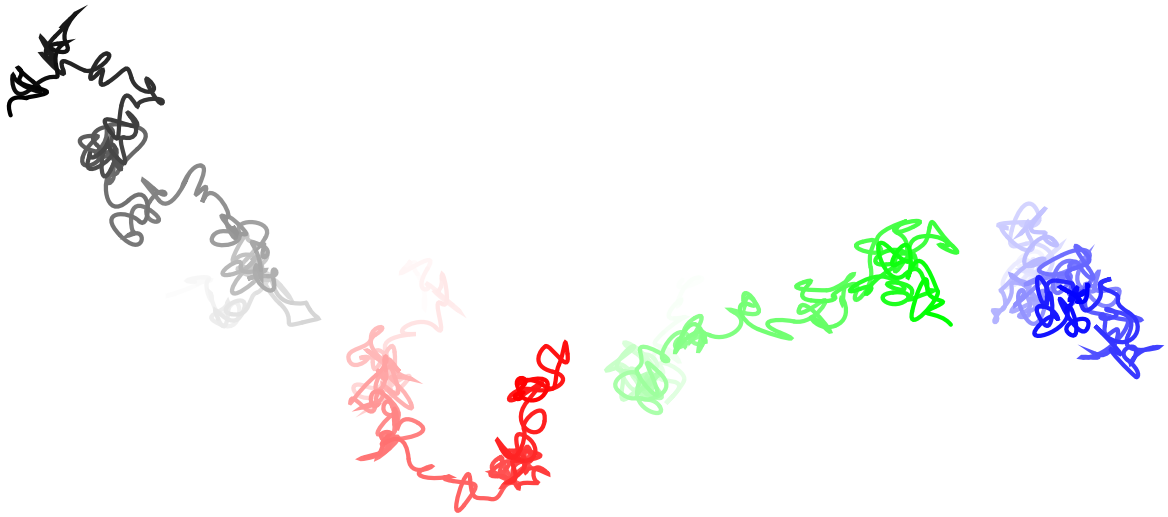
Part VIII
Mathematical and Object-Oriented Engines

by Mark Wibrow and Till Tantau

PGF comes with two useful engines: One for doing mathematics, one for doing object-oriented programming. Both engines can be used independently of the main PGF.

The job of the mathematical engine is to support mathematical operations like addition, subtraction, multiplication and division, using both integers and non-integers, but also functions such as square-roots, sine, cosine, and generate pseudo-random numbers. Mostly, you will use the mathematical facilities of PGF indirectly, namely when you write a coordinate like `(5cm*3,6cm/4)`, but the mathematical engine can also be used independently of PGF and Ti*k*Z.

The job of the object-oriented engine is to support simple object-oriented programming in TEX. It allows the definition of *classes* (without inheritance), *methods*, *attributes* and *objects*.



```
\pgfmathsetseed{1}
\foreach \col in {black,red,green,blue}
{
  \begin{tikzpicture}[x=10pt,y=10pt,ultra thick,baseline,line cap=round]
    \coordinate (current point) at (0,0);
    \coordinate (old velocity) at (0,0);
    \coordinate (new velocity) at (rand,rand);

    \foreach \i in {0,1,...,100}
    {
      \draw[\col!\i] (current point)
      .. controls ++([scale=-1]old velocity) and
                  ++(new velocity) .. ++(rand,rand)
         coordinate (current point);
      \coordinate (old velocity) at (new velocity);
      \coordinate (new velocity) at (rand,rand);
    }
  \end{tikzpicture}
}
```

93 Design Principles

PGF needs to perform many computations while typesetting a picture. For this, PGF relies on a mathematical engine, which can also be used independently of PGF, but which is distributed as part of the PGF package nevertheless. Basically, the engine provides a parsing mechanism similar to the CALC package so that expressions like `2*3cm+5cm` can be parsed; but the PGF engine is more powerful and can be extended and enhanced.

PGF provides enhanced functionality, which permits the parsing of mathematical operations involving integers and non-integers with or without units. Furthermore, various functions, including trigonometric functions and random number generators can also be parsed (see Section 94.1). The CALC macros `\setlength` and friends have PGF versions which can parse these operations and functions (see Section 94.1). Additionally, each operation and function has an independent PGF command associated with it (see Section 95), and can be accessed outside the parser.

The mathematical engine of PGF is implicitly used whenever you specify a number or dimension in a higher-level macro. For instance, you can write `\pgfpoint{2cm+4cm/2}{3cm*sin(30)}` or suchlike. However, the mathematical engine can also be used independently of the PGF core, that is, you can also just load it to get access to a mathematical parser.

93.1 Loading the Mathematical Engine

The mathematical engine of PGF is loaded automatically by PGF, but if you wish to use the mathematical engine but you do not need PGF itself, you can load the following package:

```
\usepackage{pgfmath} % LaTeX
\input{pgfmath.tex} % plain TeX
\usemodule[pgfmath]  % ConTeXt
```

This command will load the mathematical engine of PGF, but not PGF itself. It defines commands like `\pgfmathparse`.

93.2 Layers of the Mathematical Engine

Like PGF itself, the mathematical engine is also structured into different layers:

1. The top layer, which you will typically use directly, provides the command `\pgfmathparse`. This command parses a mathematical expression and evaluates it.

   Additionally, the top layer also defines some additional functions similar to the macros of the `calc` package for setting dimensions and counters. These macros are just wrappers around the `\pgfmathparse` macro.

2. The calculation layer provides macros for performing one specific computation like computing a reciprocal or a multiplication. The parser uses these macros for the actual computation.

3. The implementation layer provides the actual implementations of the computations. These can be changed (and possibly be made more efficient) without affecting the higher layers.

93.3 Efficiency and Accuracy of the Mathematical Engine

Currently, the mathematical algorithms are all implemented in TeX. This poses some intriguing programming challenges as TeX is a language for typesetting, rather than for general mathematics, and as with any programming language, there is a trade-off between accuracy and efficiency. If you find the level of accuracy insufficient for your purposes, you will have to replace the algorithms in the implementation layer.

All the fancy mathematical "bells-and-whistles" that the parser provides, come with an additional processing cost, and in some instances, such as simply setting a length to `1cm`, with no other operations involved, the additional processing time is undesirable. To overcome this, the following feature is implemented: when no mathematical operations are required, an expression can be preceded by `+`. This will bypass the parsing process and the assignment will be orders of magnitude faster. This feature *only* works with the macros for setting registers described in Section 94.1.

```
\pgfmathsetlength\mydimen{1cm}  % parsed     : slower.
\pgfmathsetlength\mydimen{+1cm} % not parsed : much faster.
```

94 Mathematical Expressions

The easiest way of using PGF's mathematical engine is to provide a mathematical expression given in familiar infix notation, for example, `1cm+4*2cm/5.5` or `2*3+3*sin(30)`. This expression can be parsed by the mathematical engine and the result can be placed in a dimension register, a counter, or a macro.

It should be noted that all calculations must not exceed ±16383.99999 at *any* point, because the underlying computations rely on TEX dimensions. This means that many of the underlying computations are necessarily approximate and, in addition, not very fast. TEX is, after all, a typesetting language and not ideally suited to relatively advanced mathematical operations. However, it is possible to change the computations as described in Section 96.

In the present section, the high-level macros for parsing an expression are explained first, then the syntax for expression is explained.

94.1 Parsing Expressions

94.1.1 Commands

The basic command for invoking the parser of PGF's mathematical engine is the following:

\pgfmathparse{⟨*expression*⟩}

This macro parses ⟨*expression*⟩ and returns the result without units in the macro `\pgfmathresult`.

Example: `\pgfmathparse{2pt+3.5pt}` will set `\pgfmathresult` to the text `5.5`.

In the following, the special properties of this command are explained. The exact syntax of mathematical expressions is explained in Sections 94.2 and 94.3.

- The result stored in the macro `\pgfmathresult` is a decimal *without units*. This is true regardless of whether the ⟨*expression*⟩ contains any unit specification. All numbers with units are converted to points first. See Section 94.1.2 for details on units.
- The parser will recognize TEX registers and box dimensions, so `\mydimen`, `0.5\mydimen`, `\wd\mybox`, `0.5\dp\mybox`, `\mycount\mydimen` and so on can be parsed.
- The ε-TEX extensions `\dimexpr`, `\numexpr`, `\glueexpr`, and `\muexpr` are recognized and evaluated. The values they result in will be used in the further evaluation, as if you had put `\the` before them.
- Parenthesis can be used to change the order of the evaluation.
- Various functions are recognized, so it is possible to parse `sin(.5*pi r)*60`, which means "the sine of 0.5 times π radians, multiplied by 60". The argument of functions can be any expression.
- Scientific notation in the form `1.234e+4` is recognized (but the restriction on the range of values still applies). The exponent symbol can be upper or lower case (i.e., `E` or `e`).
- An integer with a zero-prefix (excluding, of course zero itself), is interpreted as an octal number and is automatically converted to base 10.
- An integer with prefix `0x` or `0X` is interpreted as a hexadecimal number and is automatically converted to base 10. Alphabetic digits can be in uppercase or lowercase.
- An integer with prefix `0b` or `0B` is interpreted as a binary number and is automatically converted to base 10.
- An expression (or part of an expression) surrounded with double quotes (i.e., the character `"`) will not be evaluated. Obviously this should be used with great care.

\pgfmathqparse{⟨*expression*⟩}

This macro is similar to `\pgfmathparse`: it parses ⟨*expression*⟩ and returns the result in the macro `\pgfmathresult`. It differs in two respects. Firstly, `\pgfmathqparse` does not parse functions, scientific notation, the prefixes for binary octal, or hexadecimal numbers, nor does it accept the special use of `"`, `?` or `:` characters. Secondly, numbers in ⟨*expression*⟩ *must* specify a TEX unit (except in such instances as `0.5\pgf@x`), which greatly simplifies the problem of parsing real numbers. As a result of these restrictions `\pgfmathqparse` is about twice as fast as `\pgfmathparse`. Note that the result will still be a number without units.

\pgfmathpostparse

At the end of the parse this command is executed, allowing some custom action to be performed on the result of the parse. When this command is executed, the macro `\pgfmathresult` will hold the result of the parse (as always, without units). The result of the custom action should be used to redefine `\pgfmathresult` appropriately. By default, this command is equivalent to `\relax`. This differs from previous versions, where, if the parsed expression contained no units, the result of the parse was scaled according to the value in `\pgfmathresultunitscale` (which by default was `1`).

This scaling can be turned on again using: `\let\pgfmathpostparse=\pgfmathscaleresult`. Note, however that by scaling the result, the base conversion functions will not work, and the `"` character should not be used to quote parts of an expression.

Instead of the `\pgfmathparse` macro you can also use wrapper commands, whose usage is very similar to their cousins in the CALC package. The only difference is that the expressions can be any expression that is handled by `\pgfmathparse`. For all of the following commands, if ⟨*expression*⟩ starts with `+`, no parsing is done and a simple assignment or increment is done using normal TEX assignments or increments. This will be orders of magnitude faster than calling the parser.

The effect of the following commands is always local to the current TEX scope.

\pgfmathsetlength{⟨*register*⟩}{⟨*expression*⟩}

Basically, this command sets the length of the TEX ⟨*register*⟩ to the value specified by ⟨*expression*⟩. However, there is some fine print:

First, in case ⟨*expression*⟩ starts with a `+`, a simple TEX assignment is done. In particular, ⟨*register*⟩ can be a glue register and ⟨*expression*⟩ be something like `+1pt plus 1fil` and the ⟨*register*⟩ will be assigned the expected value.

Second, when the ⟨*expression*⟩ does not start with `+`, it is first parsed using `\pgfmathparse`, resulting in a (dimensionless) value `\pgfmathresult`. Now, if the parser encountered the unit `mu` somewhere in the expression, it assumes that ⟨*register*⟩ is a `\muskip` register and will try to assign to ⟨*register*⟩ the value `\pgfmathresult` followed by `mu`. Otherwise, in case `mu` was not encountered, it is assumed that ⟨*register*⟩ is a dimension register or a glue register and we assign `\pgfmathresult` followed by `pt` to it.

The net effect of the above is that you can write things like

13.0mu
```
\muskipdef\mymuskip=0
\pgfmathsetlength{\mymuskip}{1mu+3*4mu} \the\mymuskip
```

13.0pt
```
\dimendef\mydimen=0
\pgfmathsetlength{\mydimen}{1pt+3*4pt}  \the\mydimen
```

13.0pt
```
\skipdef\myskip=0
\pgfmathsetlength{\myskip}{1pt+3*4pt}  \the\myskip
```

One thing that will *not* work is `\pgfmathsetlength{\myskip}{1pt plus 1fil}` since the parser does not support fill's. You can, however, use the `+` notation in this case:

1.0pt plus 1.0fil
```
\skipdef\myskip=0
\pgfmathsetlength{\myskip}{+1pt plus 1fil}  \the\myskip
```

\pgfmathaddtolength{⟨*register*⟩}{⟨*expression*⟩}

Adds the value of ⟨*expression*⟩ to the TEX ⟨*register*⟩. All of the special consideration mentioned for `\pgfmathsetlength` also apply here in the same way.

\pgfmathsetcount{⟨*count register*⟩}{⟨*expression*⟩}

Sets the value of the TEX ⟨*count register*⟩, to the *truncated* value specified by ⟨*expression*⟩.

\pgfmathaddtocount{⟨*count register*⟩}{⟨*expression*⟩}

Adds the *truncated* value of ⟨*expression*⟩ to the TEX ⟨*count register*⟩.

\pgfmathsetcounter{⟨*counter*⟩}{⟨*expression*⟩}

Sets the value of the LaTeX ⟨*counter*⟩ to the *truncated* value specified by ⟨*expression*⟩.

\pgfmathaddtocounter{⟨*counter*⟩}{⟨*expression*⟩}

Adds the *truncated* value of ⟨*expression*⟩ to ⟨*counter*⟩.

\pgfmathsetmacro{⟨*macro*⟩}{⟨*expression*⟩}

Defines ⟨*macro*⟩ as the value of ⟨*expression*⟩. The result is a decimal without units.

\pgfmathsetlengthmacro{⟨*macro*⟩}{⟨*expression*⟩}

Defines ⟨*macro*⟩ as the value of ⟨*expression*⟩ *in points*.

\pgfmathtruncatemacro{⟨*macro*⟩}{⟨*expression*⟩}

Defines ⟨*macro*⟩ as the truncated value of ⟨*expression*⟩.

94.1.2 Considerations Concerning Units

As was explained earlier, the parser commands like `\pgfmathparse` will always return a result without units in it and all dimensions that have a unit like `10pt` or `1in` will first be converted to TeX points (`pt`) and, then, the unit is dropped.

Sometimes it is useful, nevertheless, to find out whether an expression or not. For this, you can use the following commands:

\ifpgfmathunitsdeclared

After a call of `\pgfmathparse` this if will be true exactly if some unit was encountered in the expression. It is always set globally in each call.

Note that *any* "mentioning" of a unit inside an expression will set this TeX-if to true. In particular, even an expressionlike `2pt/1pt`, which arguably should be considered "scalar" or "unit-free" will still have this TeX-if set to true. However, see the `scalar` function for a way to change this.

`scalar(`*value*`)`
\pgfmathscalar{*value*}

This function is the identity function on its input, but it will reset the TeX-if `\ifpgfmathunitsdeclared`. Thus, it can be used to indicate that the given ⟨*value*⟩ should be considered as a "scalar" even when it contains units; but note that it will work even when the ⟨*value*⟩ is a string or something else. The only effect of this function is to clear the unit declaration.

0.5 without unit

```
\pgfmathparse{scalar(1pt/2pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

Note, however, that this command (currently) really just clears the TeX-if as the input is scanned from left-to-right. Thus, even if there is a use of a unit before the `scalar` function is used, the TeX-if will be cleared:

2.0 without unit

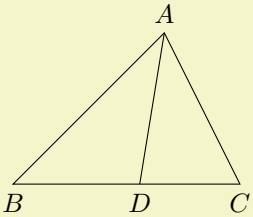
```
\pgfmathparse{1pt+scalar(1pt)} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

The other way round, a use of a unit after the `scalar` function will set the units once more.

2.0 with unit

```
\pgfmathparse{scalar(1pt)+1pt} \pgfmathresult\
\ifpgfmathunitsdeclared with \else without \fi unit
```

For these reasons, you should use the function only on the outermost level of an expression.
A typical use of this function is the following:




```
\usetikzlibrary {calc,quotes}
\tikz{
  \coordinate["$A$"]       (A) at (2,2);
  \coordinate["$B$" below] (B) at (0,0);
  \coordinate["$C$" below] (C) at (3,0);
  \draw (A) -- (B) -- (C) -- cycle;
  \path
    let \p1 =($(A)-(B)$), \p2 =($(A)-(C)$),
        \n1 = {veclen(\x1,\y1)}, \n2 = {veclen(\x2,\y2)}
    in  coordinate ["$D$" below] (D) at ($ (B)!scalar(\n1/(\n1+\n2))!(C) $);
  \draw (A) -- (D);
}
```

A special kind of units are TEX's "math units" (`mu`). It will be treated as if `pt` had been used, but you can check whether an expression contained a math unit using the following:

`\ifpgfmathmathunitsdeclared`

This TEX-if is similar to `\ifpgfmathunitsdeclared`, but it is only set when the unit `mu` is encountered at least once. In this case, `\ifpgfmathunitsdeclared` will *also* be set to true. The `scalar` function has no effect on this TEX-if.

94.2 Syntax for Mathematical Expressions: Operators

The syntax for the expressions recognized by `\pgfmathparse` and friends is rather straightforward. Let us start with the operators.

The following operators (presented in the context in which they are used) are recognized:

x `+` y (infix operator; uses the `add` function)

Adds x to y.

x `-` y (infix operator; uses the `subtract` function)

Subtracts y from x.

`-`x (prefix operator; uses the `neg` function)

Reverses the sign of x.

x `*` y (infix operator; uses the `multiply` function)

Multiplies x by y.

x `/` y (infix operator; uses the `divide` function)

Divides x by y. An error will result if y is 0, or if the result of the division is too big for the mathematical engine. Please remember when using this command that accurate (and reasonably quick) division of real numbers that are not integers is particularly tricky in TEX.

x `^` y (infix operator; uses the `pow` function)

Raises x to the power y.

x`!` (postfix operator; uses the `factorial` function)

Calculates the factorial of x.

x`r` (postfix operator; uses the `deg` function)

Converts x to degrees (x is assumed to be in radians). This operator has the same precedence as multiplication.

x `?` y `:` z (conditional operators; use the `ifthenelse` function)

`?` and `:` are special operators which can be used as a shorthand for `if` x `then` y `else` z inside the parser. The expression x is taken to be true if it evaluates to any non-zero value.

x `==` y (infix operator; uses the `equal` function)

Returns `1` if x=y, `0` otherwise.

x `>` y (infix operator; uses the `greater` function)

Returns `1` if x>y, `0` otherwise.

x `<` y (infix operator; uses the `less` function)

Returns `1` if x<y, `0` otherwise.

x `!=` y (infix operator; uses the `notequal` function)

Returns `1` if $x\neq y$, `0` otherwise.

x `>=` y (infix operator; uses the `notless` function)

Returns `1` if $x\geq y$, `0` otherwise.

x `<=` y (infix operator; uses the `notgreater` function)

Returns `1` if $x \leq y$, `0` otherwise.

x `&&` y (infix operator; uses the `and` function)

Returns `1` if both x and y evaluate to some non-zero value. Both arguments are evaluated.

x `||` y (infix operator; uses the `or` function)

Returns `1` if either x or y evaluate to some non-zero value.

`!`x (prefix operator; uses the `not` function)

Returns `1` if x evaluates to zero, `0` otherwise.

`(`x`)` (group operators)

These operators act in the usual way, that is, to control the order in which operators are executed, for example, `(1+2)*3`. This includes the grouping of arguments for functions, for example, `sin(30*10)` or `mod(72,3)` (the comma character is also treated as an operator).

Parentheses for functions with one argument are not always necessary, `sin 30` (note the space) is the same as `sin(30)`. However, functions have the highest precedence so, `sin 30*10` is the same as `sin(30)*10`.

`{`x`}` (array operators)

These operators are used to process array-like structures (within an expression these characters do not act like TEX grouping tokens). The ⟨*array specification*⟩ consists of comma separated elements, for example, `{1, 2, 3, 4, 5}`. Each element in the array will be evaluated as it is parsed, so expressions can be used. In addition, an element of an array can be an array itself, allowing multiple dimension arrays to be simulated: `{1, {2,3}, {4,5}, 6}`. When storing an array in a macro, do not forget the surrounding braces: `\def\myarray{{1,2,3}}` not `\def\myarray{1,2,3}`.

1, two, 3.0, IV, cinq, sechs, 7.0,

```
\def\myarray{{1,"two",2+1,"IV","cinq","sechs",sin(\i*5)*14}}
\foreach \i in  {0,...,6}{\pgfmathparse{\myarray[\i]}\pgfmathresult, }
```

`[`x`]` (array access operators; use the `array` function)

`[` and `]` are two operators used in one particular circumstance: to access an array (specified using the `{` and `}` operators) using the index x. Indexing starts from zero, so, if the index is greater than, or equal to, the number of values in the array, an error will occur, and zero will be returned.

-9.0

```
\def\myarray{{7,-3,4,-9,11}}
\pgfmathparse{\myarray[3]} \pgfmathresult
```

If the array is defined to have multiple dimensions, then the array access operators can be immediately repeated.

1	0	0
0	1	0
0	0	1

```
\def\print#1{\pgfmathparse{#1}\pgfmathresult}
\def\identitymatrix{{{1,0,0},{0,1,0},{0,0,1}}}
\tikz[x=0.5cm,y=0.5cm]\foreach \i in {0,1,2} \foreach \j in {0,1,2}
  \node at (\j,-\i) [anchor=base] {\print{\identitymatrix[\i][\j]}};
```

`"`x`"` (group operators)

These operators are used to quote x. However, as every expression is expanded with `\edef` before it is parsed, macros (e.g., font commands like `\tt` or `\Huge`) may need to be "protected" from this expansion (e.g., `\noexpand\Huge`). Ideally, you should avoid such macros anyway. Obviously, these operators should be used with great care as further calculations are unlikely to be possible with the result.

5 is Bigger than 0. 5 is smaller than 10.

```
\def\x{5}
\foreach \y in {0,10}{
  \pgfmathparse{\x > \y ? "\noexpand\Large Bigger" : "\noexpand\tiny smaller"}
  \x\ is \pgfmathresult\ than \y.
}
```

94.3 Syntax for Mathematical Expressions: Functions

The following functions are recognized:

abs	cosh	frac	ln	notless	sec
acos	cot	gcd	log10	oct	sign
add	deg	greater	log2	or	sin
and	depth	height	max	pi	sinh
array	dim	hex	min	pow	sqrt
asin	div	Hex	mod	rad	subtract
atan	divide	int	Mod	rand	tan
atan2	e	ifthenelse	multiply	random	tanh
bin	equal	iseven	neg	real	true
ceil	factorial	isodd	not	rnd	veclen
cos	false	isprime	notequal	round	width
cosec	floor	less	notgreater	scalar	

Each function has a PGF command associated with it (which is also shown with the function below). In general, the command is simply the name of the function prefixed with `\pgfmath`, for example, `\pgfmathadd`, but there are some notable exceptions.

94.3.1 Basic arithmetic functions

`add(`x`,`y`)`
`\pgfmathadd{`x`}{`y`}`

Adds x and y.

81.0 `\pgfmathparse{add(75,6)} \pgfmathresult`

`subtract(`x`,`y`)`
`\pgfmathsubtract{`x`}{`y`}`

Subtract y from x.

69.0 `\pgfmathparse{subtract(75,6)} \pgfmathresult`

`neg(`x`)`
`\pgfmathneg{`x`}`

This returns $-x$.

-50.0 `\pgfmathparse{neg(50)} \pgfmathresult`

`multiply(`x`,`y`)`
`\pgfmathmultiply{`x`}{`y`}`

Multiply x by y.

450.0 `\pgfmathparse{multiply(75,6)} \pgfmathresult`

`divide(`x`,`y`)`
`\pgfmathdivide{`x`}{`y`}`

Divide x by y.

12.5 `\pgfmathparse{divide(75,6)} \pgfmathresult`

`div(`x`,`y`)`
`\pgfmathdiv{`x`}{`y`}`

Divide x by y and return the integer part of the result.

8 `\pgfmathparse{div(75,9)} \pgfmathresult`

`factorial(`x`)`
`\pgfmathfactorial{`x`}`

Return $x!$.

120.0 `\pgfmathparse{factorial(5)} \pgfmathresult`

`sqrt(`x`)`
`\pgfmathsqrt{`x`}`

Calculates $\sqrt{x}$.

3.16227 `\pgfmathparse{sqrt(10)} \pgfmathresult`

93.62388 `\pgfmathparse{sqrt(8765.432)} \pgfmathresult`

`pow(`x`,`y`)`
`\pgfmathpow{`x`}{`y`}`

Raises x to the power y. For greatest accuracy, y should be an integer. If y is not an integer, the actual calculation will be an approximation of $e^{y \ln(x)}$.

128.0 `\pgfmathparse{pow(2,7)} \pgfmathresult`

`e`
`\pgfmathe`

Returns the value 2.718281828.

3.62685 `\pgfmathparse{(e^2-e^-2)/2} \pgfmathresult`

`exp(`x`)`
`\pgfmathexp{`x`}`

Maclaurin series for e^x.

2.71825 `\pgfmathparse{exp(1)} \pgfmathresult`

10.38083 `\pgfmathparse{exp(2.34)} \pgfmathresult`

`ln(`x`)`
`\pgfmathln{`x`}`

An approximation for $\ln(x)$. This uses an algorithm of Rouben Rostamian, and coefficients suggested by Alain Matthes.

2.30257 `\pgfmathparse{ln(10)} \pgfmathresult`

4.99997 `\pgfmathparse{ln(exp(5))} \pgfmathresult`

`log10(`x`)`
`\pgfmathlogten{`x`}`

An approximation for $\log_{10}(x)$.

1.99997 `\pgfmathparse{log10(100)} \pgfmathresult`

`log2(`x`)`
`\pgfmathlogtwo{`x`}`

An approximation for $\log_2(x)$.

6.99994 `\pgfmathparse{log2(128)} \pgfmathresult`

`abs(`x`)`
`\pgfmathabs{`x`}`

Evaluates the absolute value of x.

5.0 `\pgfmathparse{abs(-5)} \pgfmathresult`

-12.0 `\pgfmathparse{-abs(4*-3)} \pgfmathresult`

`mod(`x`,`y`)`
`\pgfmathmod{`x`}{`y`}`

This evaluates x modulo y, using truncated division. The sign of the result is the same as the sign of $\frac{x}{y}$.

2.0 `\pgfmathparse{mod(20,6)} \pgfmathresult`

-10.0 `\pgfmathparse{mod(-100,30)} \pgfmathresult`

`Mod(`x`,`y`)`
`\pgfmathMod{`x`}{`y`}`

This evaluates x modulo y, using floored division. The sign of the result is never negative.

20.0 `\pgfmathparse{Mod(-100,30)} \pgfmathresult`

`sign(`x`)`

\pgfmathsign{x}

Returns the sign of x.

-1 `\pgfmathparse{sign(-5)} \pgfmathresult`

0 `\pgfmathparse{sign(0)} \pgfmathresult`

1 `\pgfmathparse{sign(5)} \pgfmathresult`

94.3.2 Rounding functions

round(x)
\pgfmathround{x}

Rounds x to the nearest integer. It uses "asymmetric half-up" rounding. So `1.5` is rounded to `2`, but `-1.5` is rounded to `-2` (*not* `-1`).

2.0 `\pgfmathparse{round(32.5/17)} \pgfmathresult`

33.0 `\pgfmathparse{round(398/12)} \pgfmathresult`

floor(x)
\pgfmathfloor{x}

Rounds x down to the nearest integer.

1.0 `\pgfmathparse{floor(32.5/17)} \pgfmathresult`

33.0 `\pgfmathparse{floor(398/12)} \pgfmathresult`

-34.0 `\pgfmathparse{floor(-398/12)} \pgfmathresult`

ceil(x)
\pgfmathceil{x}

Rounds x up to the nearest integer.

2.0 `\pgfmathparse{ceil(32.5/17)} \pgfmathresult`

34.0 `\pgfmathparse{ceil(398/12)} \pgfmathresult`

-33.0 `\pgfmathparse{ceil(-398/12)} \pgfmathresult`

int(x)
\pgfmathint{x}

Returns the integer part of x.

1 `\pgfmathparse{int(32.5/17)} \pgfmathresult`

frac(*x*)
\pgfmathfrac{*x*}

Returns the fractional part of *x*.

0.91176
```
\pgfmathparse{frac(32.5/17)} \pgfmathresult
```

real(*x*)
\pgfmathreal{*x*}

Ensures *x* contains a decimal point.

4.0
```
\pgfmathparse{real(4)} \pgfmathresult
```

94.3.3 Integer arithmetic functions

gcd(*x*,*y*)
\pgfmathgcd{*x*}{*y*}

Computes the greatest common divider of *x* and *y*.

14
```
\pgfmathparse{gcd(42,56)} \pgfmathresult
```

isodd(*x*)
\pgfmathisodd{*x*}

Returns `1` if the integer part of *x* is odd. Otherwise, returns `0`.

0,1
```
\pgfmathparse{isodd(2)} \pgfmathresult,
\pgfmathparse{isodd(3)} \pgfmathresult
```

iseven(*x*)
\pgfmathiseven{*x*}

Returns `1` if the integer part of *x* is even. Otherwise, returns `0`.

1,0
```
\pgfmathparse{iseven(2)} \pgfmathresult,
\pgfmathparse{iseven(3)} \pgfmathresult
```

isprime(*x*)
\pgfmathisprime{*x*}

Returns `1` if the integer part of *x* is prime. Otherwise, returns `0`.

0,1,1,0
```
\pgfmathparse{isprime(1)} \pgfmathresult,
\pgfmathparse{isprime(2)} \pgfmathresult,
\pgfmathparse{isprime(31)} \pgfmathresult,
\pgfmathparse{isprime(64)} \pgfmathresult
```

94.3.4 Trigonometric functions

pi
\pgfmathpi

Returns the value $\pi = 3.141592654$.

3.141592654
```
\pgfmathparse{pi} \pgfmathresult
```

179.99962
```
\pgfmathparse{pi r} \pgfmathresult
```

rad(x)
\pgfmathrad{x}

Convert x to radians. x is assumed to be in degrees.

1.57079
```
\pgfmathparse{rad(90)} \pgfmathresult
```

deg(x)
\pgfmathdeg{x}

Convert x to degrees. x is assumed to be in radians.

269.999
```
\pgfmathparse{deg(3*pi/2)} \pgfmathresult
```

sin(x)
\pgfmathsin{x}

Sine of x. By employing the `r` operator, x can be in radians.

0.86603
```
\pgfmathparse{sin(60)} \pgfmathresult
```

0.86601
```
\pgfmathparse{sin(pi/3 r)} \pgfmathresult
```

cos(x)
\pgfmathcos{x}

Cosine of x. By employing the `r` operator, x can be in radians.

0.5
```
\pgfmathparse{cos(60)} \pgfmathresult
```

0.49998
```
\pgfmathparse{cos(pi/3 r)} \pgfmathresult
```

tan(x)
\pgfmathtan{x}

Tangent of x. By employing the `r` operator, x can be in radians.

1.00005
```
\pgfmathparse{tan(45)} \pgfmathresult
```

1.0
```
\pgfmathparse{tan(2*pi/8 r)} \pgfmathresult
```

sec(x)
\pgfmathsec{x}

Secant of x. By employing the `r` operator, x can be in radians.

1.41429
```
\pgfmathparse{sec(45)} \pgfmathresult
```

cosec(x)
\pgfmathcosec{x}

Cosecant of x. By employing the `r` operator, x can be in radians.

2.0
```
\pgfmathparse{cosec(30)} \pgfmathresult
```

`cot(`x`)`
`\pgfmathcot{`x`}`

Cotangent of x. By employing the `r` operator, x can be in radians.

3.73215
```
\pgfmathparse{cot(15)} \pgfmathresult
```

`asin(`x`)`
`\pgfmathasin{`x`}`

Arcsine of x. The result is in degrees and in the range $\pm 90°$.

44.99945
```
\pgfmathparse{asin(0.7071)} \pgfmathresult
```

`acos(`x`)`
`\pgfmathacos{`x`}`

Arccosine of x in degrees. The result is in the range $[0°, 180°]$.

60.0
```
\pgfmathparse{acos(0.5)} \pgfmathresult
```

`atan(`x`)`
`\pgfmathatan{`x`}`

Arctangent of x in degrees.

45.0
```
\pgfmathparse{atan(1)} \pgfmathresult
```

`atan2(`y`,`x`)`
`\pgfmathatantwo{`y`}{`x`}`

Arctangent of $y \div x$ in degrees. This also takes into account the quadrants.

-53.13011
```
\pgfmathparse{atan2(-4,3)} \pgfmathresult
```

`/pgf/trig format=deg|rad` (no default, initially `deg`)

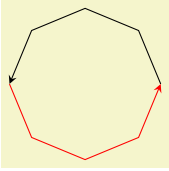
Allows to define whether trigonometric math functions (i.e. all in this subsection) operate with degrees or with radians.

0.7071
```
\pgfmathparse{cos(45)} \pgfmathresult
```

0.0
```
\pgfkeys{/pgf/trig format=rad}
\pgfmathparse{cos(pi/2)} \pgfmathresult
```

The initial configuration `trig format=deg` is the base of PGF: almost all of it is based on degrees.

Specifying `trig format=rad` is most useful for data visualization where the angles are typically given in radians. However, it is applied to all trigonometric functions for which the option applies, including any drawing instructions which operate on angles.



```
\begin{tikzpicture}
    \draw[-stealth]
        (0:1) -- (45:1) -- (90:1) -- (135:1) -- (180:1);

    \draw[-stealth,trig format=rad,red]
        (pi:1) -- (5/4*pi:1) -- (6/4*pi:1) -- (7/4*pi:1) -- (2*pi:1);
\end{tikzpicture}
```

Warning: At the time of this writing, this feature is "experimental". Please handle it with care: there may be path instructions or libraries in PGF which rely on `trig format=deg`. The intended usage of `trig format=rad` is for local scopes – and as option for data visualization.

94.3.5 Comparison and logical functions

`equal(`x`,`y`)`
`\pgfmathequal{`x`}{`y`}`

This returns `1` if $x = y$ and `0` otherwise.

```
1  \pgfmathparse{equal(20,20)} \pgfmathresult
```

`greater(`x`,`y`)`
`\pgfmathgreater{`x`}{`y`}`

This returns `1` if $x > y$ and `0` otherwise.

```
0  \pgfmathparse{greater(20,25)} \pgfmathresult
```

`less(`x`,`y`)`
`\pgfmathless{`x`}{`y`}`

This returns `1` if $x < y$ and `0` otherwise.

```
1  \pgfmathparse{less(20,25)} \pgfmathresult
```

`notequal(`x`,`y`)`
`\pgfmathnotequal{`x`}{`y`}`

This returns `0` if $x = y$ and `1` otherwise.

```
1  \pgfmathparse{notequal(20,25)} \pgfmathresult
```

`notgreater(`x`,`y`)`
`\pgfmathnotgreater{`x`}{`y`}`

This returns `1` if $x \leq y$ and `0` otherwise.

```
1  \pgfmathparse{notgreater(20,25)} \pgfmathresult
```

`notless(`x`,`y`)`
`\pgfmathnotless{`x`}{`y`}`

This returns `1` if $x \geq y$ and `0` otherwise.

```
0  \pgfmathparse{notless(20,25)} \pgfmathresult
```

`and(`x`,`y`)`
`\pgfmathand{`x`}{`y`}`

This returns `1` if x and y both evaluate to non-zero values. Otherwise `0` is returned.

```
0  \pgfmathparse{and(5>4,6>7)} \pgfmathresult
```

`or(`x`,`y`)`

\pgfmathor{x}{y}

This returns 1 if either x or y evaluate to non-zero values. Otherwise 0 is returned.

1
```
\pgfmathparse{or(5>4,6>7)} \pgfmathresult
```

not(x)
\pgfmathnot{x}

This returns 1 if $x = 0$, otherwise 0.

0
```
\pgfmathparse{not(true)} \pgfmathresult
```

ifthenelse(x,y,z)
\pgfmathifthenelse{x}{y}{z}

This returns y if x evaluates to some non-zero value, otherwise z is returned.

no
```
\pgfmathparse{ifthenelse(5==4,"yes","no")} \pgfmathresult
```

true
\pgfmathtrue

This evaluates to 1.

yes
```
\pgfmathparse{true ? "yes" : "no"} \pgfmathresult
```

false
\pgfmathfalse

This evaluates to 0.

no
```
\pgfmathparse{false ? "yes" : "no"} \pgfmathresult
```

94.3.6 Pseudo-random functions

rnd
\pgfmathrnd

Generates a pseudo-random number between 0 and 1 with a uniform distribution.

0.69621, 0.10826, 0.8601, 0.21024, 0.60327, 0.39178, 0.697, 0.26112, 0.38905, 0.69016,

```
\foreach \x in {1,...,10}{\pgfmathparse{rnd}\pgfmathresult, }
```

rand
\pgfmathrand

Generates a pseudo-random number between -1 and 1 with a uniform distribution.

-0.30379, 0.13586, 0.91423, 0.2203, -0.30508, -0.54727, -0.2593, -0.69032, 0.47404, -0.2289,

```
\foreach \x in {1,...,10}{\pgfmathparse{rand}\pgfmathresult, }
```

random(x,y)

\pgfmathrandom{x,y}

This function takes zero, one or two arguments. If there are zero arguments, a uniform random number between 0 and 1 is generated. If there is one argument x, a random integer between 1 and x is generated. Finally, if there are two arguments, a random integer between x and y is generated. If there are no arguments, the PGF command should be called as follows: `\pgfmathrandom{}`.

0.69621, 0.10826, 0.8601, 0.21024, 0.60327, 0.39178, 0.697, 0.26112, 0.38905, 0.69016,

```
\foreach \x in {1,...,10}{\pgfmathparse{random()}\pgfmathresult, }
```

22, 48, 35, 38, 56, 69, 42, 23, 3, 8,

```
\foreach \x in {1,...,10}{\pgfmathparse{random(100)}\pgfmathresult, }
```

292, 302, 294, 460, 699, 385, 635, 602, 708, 408,

```
\foreach \x in {1,...,10}{\pgfmathparse{random(232,762)}\pgfmathresult, }
```

94.3.7 Base conversion functions

hex(x)
\pgfmathhex{x}

Convert x (assumed to be an integer in base 10) to a hexadecimal representation, using lower case alphabetic digits. No further calculation will be possible with the result.

ffff `\pgfmathparse{hex(65535)} \pgfmathresult`

Hex(x)
\pgfmathHex{x}

Convert x (assumed to be an integer in base 10) to a hexadecimal representation, using upper case alphabetic digits. No further calculation will be possible with the result.

FFFF `\pgfmathparse{Hex(65535)} \pgfmathresult`

oct(x)
\pgfmathoct{x}

Convert x (assumed to be an integer in base 10) to an octal representation. No further calculation should be attempted with the result, as the parser can only process numbers converted to base 10.

77 `\pgfmathparse{oct(63)} \pgfmathresult`

bin(x)
\pgfmathbin{x}

Convert x (assumed to be an integer in base 10) to a binary representation. No further calculation should be attempted with the result, as the parser can only process numbers converted to base 10.

10111001 `\pgfmathparse{bin(185)} \pgfmathresult`

94.3.8 Miscellaneous functions

`min`(x_1,x_2,…,x_n)
`\pgfmathmin`{x_1,x_2,…}{…,x_{n-1},x_n}

Return the minimum value from $x_1 \ldots x_n$. For historical reasons, the command `\pgfmathmin` takes two arguments, but each of these can contain an arbitrary number of comma separated values.

-8.0 `\pgfmathparse{min(3,4,-2,250,-8,100)} \pgfmathresult`

`max`(x_1,x_2,…,x_n)
`\pgfmathmax`{x_1,x_2,…}{…,x_{n-1},x_n}

Return the maximum value from $x_1 \ldots x_n$. Again, for historical reasons, the command `\pgfmathmax` takes two arguments, but each of these can contain an arbitrary number of comma separated values.

250.0 `\pgfmathparse{max(3,4,-2,250,-8,100)} \pgfmathresult`

`veclen`(x,y)
`\pgfmathveclen`{x}{y}

Calculates $\sqrt{(x^2+y^2)}$. This uses a polynomial approximation, based on ideas of Rouben Rostamian

12.99976 `\pgfmathparse{veclen(12,5)} \pgfmathresult`

`array`(x,y)
`\pgfmatharray`{x}{y}

This accesses the array x at the index y. The array must begin and end with braces (e.g., `{1,2,3,4}`) and array indexing starts at `0`.

17 `\pgfmathparse{array({9,13,17,21},2)} \pgfmathresult`

`dim`(x)
`\pgfmathdim`{x}

This returns the dimension of the array x. The array must begin and end with braces (e.g., `{1,2,3,4}`).

4 `\pgfmathparse{dim({1,2,3,4})} \pgfmathresult`

The following hyperbolic functions were adapted from code suggested by Martin Heller:

`sinh`(x)
`\pgfmathsinh`{x}

The hyperbolic sine of x

0.52103 `\pgfmathparse{sinh(0.5)} \pgfmathresult`

`cosh`(x)
`\pgfmathcosh`{x}

The hyperbolic cosine of x

1.12767 `\pgfmathparse{cosh(0.5)} \pgfmathresult`

`tanh(`x`)`
`\pgfmathtanh{`x`}`

The hyperbolic tangent of x

0.462
```
\pgfmathparse{tanh(0.5)} \pgfmathresult
```

`width("`x`")`
`\pgfmathwidth{"`x`"}`

Return the width of a TEX (horizontal) box containing x. The quote characters are necessary to prevent x from being parsed. It is important to remember that any expression is expanded with `\edef` before being parsed, so any macros (e.g., font commands like `\tt` or `\Huge`) will need to be "protected" (e.g., `\noexpand\Huge` is usually sufficient).

78.46
```
\pgfmathparse{width("Some Lovely Text")} \pgfmathresult
```

Note that results of this method are provided in points.

`height("`x`")`
`\pgfmathheight{"`x`"}`

Return the height of a box containing x.

7.05
```
\pgfmathparse{height("Some Lovely Text")} \pgfmathresult
```

`depth("`x`")`
`\pgfmathdepth{"`x`"}`

Returns the depth of a box containing x.

2.05
```
\pgfmathparse{depth("Some Lovely Text")} \pgfmathresult
```

95 Additional Mathematical Commands

Instead of parsing and evaluating complex expressions, you can also use the mathematical engine to evaluate a single mathematical operation. The macros used for many of these computations are listed above in Section 94.3. PGF also provides some additional commands which are shown below:

95.1 Basic arithmetic functions

In addition to the commands described in Section 94.3.1, the following command is provided:

\pgfmathreciprocal{⟨*x*⟩}

Defines \pgfmathresult as $1 \div \langle x \rangle$. This provides greatest accuracy when x is small.

95.2 Comparison and logical functions

In addition to the commands described in Section 94.3.5, the following command was provided by Christian Feuersänger:

\pgfmathapproxequalto{⟨*x*⟩}{⟨*y*⟩}

Defines \pgfmathresult 1.0 if $|\langle x \rangle - \langle y \rangle| < 0.0001$, but 0.0 otherwise. As a side-effect, the global boolean \ifpgfmathcomparison will be set accordingly.

95.3 Pseudo-Random Numbers

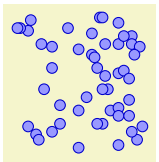
In addition to the commands described in Section 94.3.6, the following commands are provided:

\pgfmathgeneratepseudorandomnumber

Defines \pgfmathresult as a pseudo-random integer between 1 and $2^{31} - 1$. This uses a linear congruency generator, based on ideas of Erich Janka.

\pgfmathrandominteger{⟨*macro*⟩}{⟨*minimum*⟩}{⟨*maximum*⟩}

This defines ⟨*macro*⟩ as a pseudo-randomly generated integer from the range ⟨*minimum*⟩ to ⟨*maximum*⟩ (inclusive).



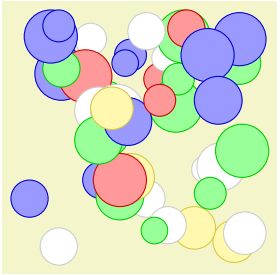
```
\begin{pgfpicture}
  \foreach \x in {1,...,50}{
    \pgfmathrandominteger{\a}{1}{50}
    \pgfmathrandominteger{\b}{1}{50}
    \pgfpathcircle{\pgfpoint{+\a pt}{+\b pt}}{+2pt}
    \color{blue!40!white}
    \pgfsetstrokecolor{blue!80!black}
    \pgfusepath{stroke, fill}
  }
\end{pgfpicture}
```

\pgfmathdeclarerandomlist{⟨*list name*⟩}{{⟨*item-1*⟩}{⟨*item-2*⟩}...}

This creates a list of items with the name ⟨*list name*⟩.

\pgfmathrandomitem{⟨*macro*⟩}{⟨*list name*⟩}

Select an item from a random list ⟨*list name*⟩. The selected item is placed in ⟨*macro*⟩.



```
\begin{pgfpicture}
  \pgfmathdeclarerandomlist{color}{{red}{blue}{green}{yellow}{white}}
  \foreach \a in {1,...,50}{
    \pgfmathrandominteger{\x}{1}{85}
    \pgfmathrandominteger{\y}{1}{85}
    \pgfmathrandominteger{\r}{5}{10}
    \pgfmathrandomitem{\c}{color}
    \pgfpathcircle{\pgfpoint{+\x pt}{+\y pt}}{+\r pt}
    \color{\c!40!white}
    \pgfsetstrokecolor{\c!80!black}
    \pgfusepath{stroke, fill}
  }
\end{pgfpicture}
```

\pgfmathsetseed{⟨*integer*⟩}

Explicitly sets the seed for the pseudo-random number generator. By default it is set to the value of \time×\year.

95.4 Base Conversion

PGF provides limited support for conversion between *representations* of numbers. Currently the numbers must be positive integers in the range 0 to $2^{31}-1$, and the bases in the range 2 to 36. All digits representing numbers greater than 9 (in base ten), are alphabetic, but may be upper or lower case.

In addition to the commands described in Section 94.3.7, the following commands are provided:

\pgfmathbasetodec{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base*⟩ to base 10. Alphabetic digits can be upper or lower case.

```
\pgfmathbasetodec\mynumber{107f}{16} \mynumber
```
4223

Note that, as usual in TEX, the braces around an argument can be omitted if the argument is just a single token (a macro name is a single token).

```
\pgfmathbasetodec\mynumber{33FC}{20} \mynumber
```
25512

\pgfmathdectobase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base 10 to base ⟨*base*⟩. Any resulting alphabetic digits are in *lower case*.

```
\pgfmathdectobase\mynumber{65535}{16} \mynumber
```
ffff

\pgfmathdectoBase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base*⟩}

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base 10 to base ⟨*base*⟩. Any resulting alphabetic digits are in *upper case*.

```
\pgfmathdectoBase\mynumber{65535}{16} \mynumber
```
FFFF

\pgfmathbasetobase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base-1*⟩}{⟨*base-2*⟩}

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base-1*⟩ to base ⟨*base-2*⟩. Alphabetic digits in ⟨*number*⟩ can be upper or lower case, but any resulting alphabetic digits are in *lower case*.

```
\pgfmathbasetobase\mynumber{11011011}{2}{16} \mynumber
```
db

\pgfmathbasetoBase{⟨*macro*⟩}{⟨*number*⟩}{⟨*base-1*⟩}{⟨*base-2*⟩}

Defines ⟨*macro*⟩ as the result of converting ⟨*number*⟩ from base ⟨*base-1*⟩ to base ⟨*base-2*⟩. Alphabetic digits in ⟨*number*⟩ can be upper or lower case, but any resulting alphabetic digits are in *upper case*.

```
\pgfmathbasetoBase\mynumber{121212}{3}{12} \mynumber
```
31B

\pgfmathsetbasenumberlength{⟨*integer*⟩}

Sets the number of digits in the result of a base conversion to ⟨*integer*⟩. If the result of a conversion has less digits than this number, it is prefixed with zeros.

```
\pgfmathsetbasenumberlength{8}
\pgfmathdectobase\mynumber{15}{2} \mynumber
```
00001111

\pgfmathtodigitlist{⟨*macro*⟩}{⟨*number*⟩}

This command converts ⟨*number*⟩ into a comma-separated list of digits and stores the result in ⟨*macro*⟩. The {⟨*number*⟩} is *not* parsed before processing.



```
\pgfmathsetbasenumberlength{8}
\begin{tikzpicture}[x=0.25cm, y=0.25cm]
  \foreach \n [count=\y] in {0, 60, 102, 102, 126, 102, 102, 102, 0}{
    \pgfmathdectobase{\binary}{\n}{2}
    \pgfmathtodigitlist{\digitlist}{\binary}
    \foreach \digit [count=\x, evaluate={\c=\digit*50+15;}] in \digitlist
      \fill [fill=black!\c] (\x, -\y) rectangle ++(1,1);
  }
\end{tikzpicture}
```

95.5 Angle Computations

Unlike the rest of the math engine, which is a "standalone" package, the following commands only work in conjunction with the core of PGF.

\pgfmathanglebetweenpoints{⟨p⟩}{⟨q⟩}

Returns the angle of a line from ⟨p⟩ to ⟨q⟩ relative to a line going straight right from ⟨p⟩.

45.0

```
\pgfmathanglebetweenpoints{\pgfpoint{1cm}{3cm}}{\pgfpoint{2cm}{4cm}}
\pgfmathresult
```

\pgfmathanglebetweenlines{⟨p_1⟩}{⟨q_1⟩}{⟨p_2⟩}{⟨q_2⟩}

Returns the clockwise angle between a line going through p_1 and q_1 and a line going through p_2 and q_2.

270.0

```
\pgfmathanglebetweenlines{\pgfpoint{1cm}{3cm}}{\pgfpoint{2cm}{4cm}}
                         {\pgfpoint{0cm}{1cm}}{\pgfpoint{1cm}{0cm}}
\pgfmathresult
```

96 Customizing the Mathematical Engine

Perhaps you have a desire for some function that PGF does not provide. Perhaps you are not happy with the accuracy or efficiency of some of the algorithms that are implemented in PGF. In these cases you will want to add a function to the parser or replace the current implementations of the algorithms with your own code.

The mathematical engine was designed with such customization in mind. It is possible to add new functions, or modify the code for existing functions. Note, however, that whilst adding new operators is possible, it can be a rather tricky business and is only recommended for adventurous users.

To add a new function to the math engine the following command can be used:

`\pgfmathdeclarefunction*{`⟨*function name*⟩`}{`⟨*number of arguments*⟩`}{`⟨*code*⟩`}`

This will set up the parser to recognize a function called ⟨*name*⟩. The name of the function can consist of, uppercase or lowercase letters, numbers or the underscore `_`. In line with many programming languages, a function name cannot begin with a number or contain any spaces. The function may not have been declared earlier, unless the optional star (`*`) is provided, which forces an "overwriting" of the function by the new function. Note that you *should never change the arity of standard functions* and you should normally use `\pgfmathredeclarefunction`, which does not allow you to do anything wrong here.

The ⟨*number of arguments*⟩ can be any positive integer, zero, or the value `...`, which indicates a variable number of arguments. PGF treats constants, such as `pi` and `e`, as functions with zero arguments. Functions with more than nine arguments or with a variable number of arguments are a "bit special" and are discussed below.

The effect of ⟨*code*⟩ should be to set the macro `\pgfmathresult` to the correct value (namely to the result of the computation without units). Furthermore, the function should have no other side effects, that is, it should not change any global values. As an example, consider the creation of a new function `double`, which takes one argument, and returns the value of that argument times two.

88.6
```
\makeatletter
\pgfmathdeclarefunction{double}{1}{
  \begingroup
    \pgf@x=#1pt\relax
    \multiply\pgf@x by2\relax
    \pgfmathreturn\pgf@x
  \endgroup
}
\makeatother
\pgfmathparse{double(44.3)}\pgfmathresult
```

The macro `\pgfmathreturn`⟨*tokens*⟩ must be directly followed by an `\endgroup` and will save the result of the computation, by defining `\pgfmathresult` as the expansion of ⟨*tokens*⟩ (without units) outside the group, so ⟨*tokens*⟩ must be something that can be assigned to a dimension register.

Alternatively, the `\pgfmathsmuggle`⟨*macro*⟩ can be used. This must also be directly followed by an `\endgroup` and will simply "smuggle" the definition of ⟨*macro*⟩ outside the TEX-group.

By performing computations within a TEX-group, PGF registers such as `\pgf@x`, `\pgf@y` and `\c@pgf@counta`, `\c@pgfcountb`, and so forth, can be used at will.

Beyond setting up the parser, this command also defines two macros which provide access to the function independently of the parser:

- `\pgfmath`⟨*function name*⟩
  This macro will provide a "public" interface for the function ⟨*function name*⟩ allowing the function to be called independently of the parser. All arguments passed to this macro are evaluated using `\pgfmathparse` and then passed on to the following macro:
- `\pgfmath`⟨*function name*⟩`@`
  This macro is the "private" implementation of the function's algorithm (but note that, for speed, the parser calls this macro rather than the "public" one). Arguments passed to this macro are expected to be numbers without units. It is defined using ⟨*code*⟩, but need not be self-contained.

For functions that are declared with less than ten arguments, the public macro is defined in the same way as normal TEX macros using, for example, `\def\pgfmathNoArgs{`⟨*code*⟩`}` for a function with no arguments, or `\def\pgfmathThreeArgs#1#2#3{`⟨*code*⟩`}` for a function with three arguments. The private

macro is defined in the same way, and each argument can therefore be accessed in ⟨*code*⟩ using `#1`, `#2` and so on.

For functions with more than nine arguments, or functions with a variable number of arguments, these macros are only defined as taking *one* argument. The public macro expects its arguments to be comma separated, for example, `\pgfmathVariableArgs{1.1,3.5,-1.5,2.6}`. Each argument is parsed and passed on to the private macro as follows: `\pgfmathVariableArgs@{{1.1}{3.5}{-1.5}{2.6}}`. This means that some "extra work" will be required to access each argument (although it is a fairly simple task).

Note that there are two exceptions to this arrangement: the public versions of the `min` and `max` functions still take two arguments for compatibility with older versions, but each of these arguments can take several comma separated values.

To redefine a function use the following command:

`\pgfmathredeclarefunction{`⟨*function name*⟩`}{`⟨*code*⟩`}`

This command redefines the `\pgfmath`⟨*function name*⟩`@` macro with the new ⟨*code*⟩. See the description of the `\pgfmathdeclarefunction` for details. You cannot change the number of arguments for an existing function.

84.0 126.0

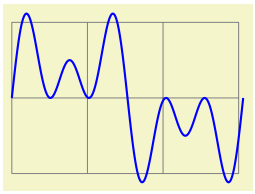
```
\makeatletter
\pgfmathdeclarefunction{foo}{1}{
  \begingroup
    \pgf@x=#1pt\relax
    \multiply\pgf@x by2\relax
    \pgfmathreturn\pgf@x
  \endgroup
}
\pgfmathparse{foo(42)}\pgfmathresult
\pgfmathredeclarefunction{foo}{
  \begingroup
    \pgf@x=#1pt\relax
    \multiply\pgf@x by3\relax
    \pgfmathreturn\pgf@x
  \endgroup
}
\pgfmathparse{foo(42)}\pgfmathresult
\makeatother
```

PGF uses the last known definition of a function within the prevailing scope, so it is possible for a function to be redefined locally. You should also remember that any `.sty` or `.tex` file containing any re-implementations should be loaded after `pgfmath`.

In addition to the above commands, the following key is provided to quickly create simple ad hoc functions which can greatly improve the readability of code, and is particularly useful in Ti*k*Z:

`/pgf/declare function=`⟨*function definitions*⟩ (no default)

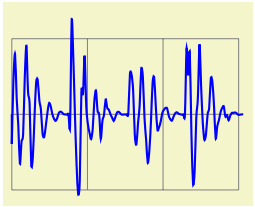
This key allows simple functions to be created locally. Its use is perhaps best illustrated by an example:



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm,
    declare function={
      sines(\t,\a,\b)=1 + 0.5*(sin(\t)+sin(\t*\a)+sin(\t*\b));
    }]
    plot [domain=0:360, samples=144, smooth] (\x,{sines(\x,3,5)});
\end{tikzpicture}
```

Each definition in ⟨*function definitions*⟩ takes the form ⟨*name*⟩`(`⟨*arguments*⟩`)=`⟨*definition*⟩`;` (note the semicolon at the end, this is very important). If multiple functions are being defined, the semicolon is used to separate them (*not* a comma). The function ⟨*name*⟩ can be any name that is not already a function name in the current scope. The list of ⟨*arguments*⟩ are commands such as `\x`, or `\y` (it is not possible to declare functions that take variable numbers of arguments using this key). If the function takes no arguments, then the parentheses need not be used. The ⟨*definition*⟩ should be an expression that can be parsed by the mathematical engine and should use the commands specified in ⟨*arguments*⟩.

When specifying multiple functions, functions that appear later on in ⟨*function definitions*⟩ can refer to earlier functions:



```
\begin{tikzpicture}[
  declare function={
    excitation(\t,\w) = sin(\t*\w);
    noise             = rnd - 0.5;
    source(\t)        = excitation(\t,20) + noise;
    filter(\t)        = 1 - abs(sin(mod(\t, 90)));
    speech(\t)        = 1 + source(\t)*filter(\t);
  }
]
  \draw [help lines] (0,0) grid (3,2);
  \draw [blue, thick, x=0.0085cm, y=1cm] (0,1) --
    plot [domain=0:360, samples=144, smooth] (\x,{speech(\x)});
\end{tikzpicture}
```

`/pgf/declare function/execute at begin function`=⟨*tokens*⟩ (no default)

These ⟨*tokens*⟩ are inserted just before `\pgfmathdeclarefunction` scans the body of the function definition. This is a rather low-level option, so you should read the implementation to figure out where the ⟨*tokens*⟩ are inserted.

`/pgf/declare function/execute at end function`=⟨*tokens*⟩ (no default)

These ⟨*tokens*⟩ are inserted just after `\pgfmathdeclarefunction` has finished scanning the body of the function definition. This is a rather low-level option, so you should read the implementation to figure out where the ⟨*tokens*⟩ are inserted.

`/pgf/declare function/ignore spaces`=⟨*boolean*⟩ (no default)

Uses the two previously described keys `/pgf/declare function/execute at begin function` and `/pgf/declare function/execute at end function` to install catcodes such that spaces inside the body of the function definition of `\pgfmathdeclarefunction` are ignored. The usual TEX tokenization rules apply, so if the body of the function had already been tokenized by other means this will become ineffective. If you want to use a space you can use `~` in the function body which has its catcode set to 10 (space).

97 Number Printing

An extension by Christian Feuersänger

PGF supports number printing in different styles and rounds to arbitrary precision.

`\pgfmathprintnumber{`⟨*x*⟩`}`

Generates pretty-printed output for the (real) number ⟨*x*⟩. The input number ⟨*x*⟩ is parsed using `\pgfmathfloatparsenumber` which allows arbitrary precision.

Numbers are typeset in math mode using the current set of number printing options, see below. Optional arguments can also be provided using `\pgfmathprintnumber[`⟨*options*⟩`]`⟨*x*⟩.

`\pgfmathprintnumberto{`⟨*x*⟩`}{`⟨*macro*⟩`}`

Returns the resulting number into ⟨*macro*⟩ instead of typesetting it directly.

`/pgf/number format/fixed` (no value)

Configures `\pgfmathprintnumber` to round the number to a fixed number of digits after the period, discarding any trailing zeros.

4.57 0 0.1 24,415.98 123,456.12

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 97.1 for how to change the appearance.

`/pgf/number format/fixed zerofill=`{⟨*boolean*⟩} (default `true`)

Enables or disables zero filling for any number drawn in fixed point format.

4.57 0.00 0.10 24,415.98 123,456.12

```
\pgfkeys{/pgf/number format/.cd,fixed,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

This key affects numbers drawn with `fixed` or `std` styles (the latter only if no scientific format is chosen).

4.57 $5 \cdot 10^{-5}$ 1.00 $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,std,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-05}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 97.1 for how to change the appearance.

`/pgf/number format/sci` (no value)

Configures `\pgfmathprintnumber` to display numbers in scientific format, that means sign, mantissa and exponent (basis 10). The mantissa is rounded to the desired `precision` (or `sci precision`, see below).

$4.57 \cdot 10^{0}$ $5 \cdot 10^{-4}$ $1 \cdot 10^{-1}$ $2.44 \cdot 10^{4}$ $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,sci,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See section 97.1 for how to change the exponential display style.

`/pgf/number format/sci zerofill=`{⟨*boolean*⟩} (default `true`)

Enables or disables zero filling for any number drawn in scientific format.

$4.57 \cdot 10^{0}$ $5.00 \cdot 10^{-4}$ $1.00 \cdot 10^{-1}$ $2.44 \cdot 10^{4}$ $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,sci,sci zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

As with `fixed zerofill`, this option does only affect numbers drawn in `sci` format (or `std` if the scientific format is chosen).

See section 97.1 for how to change the exponential display style.

`/pgf/number format/zerofill=`{⟨*boolean*⟩} (style, default `true`)

Sets both `fixed zerofill` and `sci zerofill` at once.

`/pgf/number format/std` (no value)
`/pgf/number format/std=`⟨*lower e*⟩ (no default)
`/pgf/number format/std=`⟨*lower e*⟩`:`⟨*upper e*⟩ (no default)

Configures `\pgfmathprintnumber` to a standard algorithm. It chooses either `fixed` or `sci`, depending on the order of magnitude. Let $n = s \cdot m \cdot 10^{e}$ be the input number and p the current precision. If $-p/2 \leq e \leq 4$, the number is displayed using `fixed` format. Otherwise, it is displayed using `sci` format.

4.57 $5 \cdot 10^{-4}$ 0.1 24,415.98 $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,std,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

The parameters can be customized using the optional integer argument(s): if ⟨*lower e*⟩ $\leq e \leq$ ⟨*upper e*⟩, the number is displayed in `fixed` format, otherwise in `sci` format. Note that ⟨*lower e*⟩ should be negative for useful results. The precision used for the scientific format can be adjusted with `sci precision` if necessary.

`/pgf/number format/relative*=`⟨*exponent base 10*⟩ (no default)

Configures `\pgfmathprintnumber` to format numbers relative to an order of magnitude, 10^{r}, where r is an integer number.

This key addresses different use-cases.

First use-case: provide a unified format for a *sequence* of numbers. Consider the following test:

0 1.2 6 20.6 87

```
\pgfkeys{/pgf/number format/relative*={1}}
\pgfmathprintnumber{6.42e-16}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{6}\hspace{1em}
\pgfmathprintnumber{20.6}\hspace{1em}
\pgfmathprintnumber{87}
```

With any other style, the `6.42e-16` would have been formatted as an isolated number. Here, it is rounded to `0` because when viewed relative to 10^1 (the exponent 1 is the argument for `relative`), it has no significant digits.

123 0 0

```
\pgfkeys{/pgf/number format/relative*={2}}
\pgfmathprintnumber{123.345}\hspace{1em}
\pgfmathprintnumber{0.0012}\hspace{1em}
\pgfmathprintnumber{0.0014}\hspace{1em}
```

The example above applies the initial `precision=2` to `123.345` – relative to 100. Two significant digits of `123.345` relative to 100 are `123`. Note that the "2 significant digits of `123.345`" translates to "round `1.2345` to 2 digits", which would yield `1.2300`. Similarly, the other two numbers are `0` compared to 100 using the given `precision`.

123.345 $1.2 \cdot 10^{-3}$ $1.4 \cdot 10^{-3}$

```
\pgfkeys{/pgf/number format/relative*={-3}}
\pgfmathprintnumber{123.345}\hspace{1em}
\pgfmathprintnumber{0.0012}\hspace{1em}
\pgfmathprintnumber{0.0014}\hspace{1em}
```

Second use-case: improve rounding in the presence of *inaccurate* numbers. Let us suppose that some limited-precision arithmetic resulted in the result `123456999` (like the `fpu` of PGF). You know that its precision is about five or six significant digits. And you want to provide a fixed point output. In this case, the trailing digits `....999` are a numerical artifact due to the limited precision. Use `relative*=3,precision=0` to eliminate the artifacts:

$1.23457 \cdot 10^8$ $1.23457 \cdot 10^8$

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

Here, `precision=0` means that we inspect `123456.999` and round that number to 0 digits. Finally, we move the period back to its initial position. Adding `relative style=fixed` results in fixed point output format:

123,457,000 123,457,000

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0,relative style=fixed}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

Note that there is another alternative for this use-case which is discussed later: the `fixed relative` style.

123,457,000 123,457,000

```
\pgfkeys{/pgf/number format/.cd,fixed relative,precision=6}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

You might wonder why there is an asterisk in the key's name. The short answer is: there is also a `/pgf/number format/relative` number printer which does unexpected things. The key `relative*` repairs this. Existing code will still use the old behavior.

Technically, the key works as follows: as already explained above, `relative*=3` key applied to `123456999.12` moves the period by three positions and analyzes `123456.99912`. Mathematically speaking, we are given a number $x = \pm m \cdot 10^e$ and we attempt to apply `relative*=`r. The method then rounds $x/10^r$ to `precision` digits. Afterwards, it multiplies the result by 10^r and typesets it.

`/pgf/number format/every relative` (style, no value)

A style which configures how the `relative` method finally displays its results.

The initial configuration is

```
\pgfkeys{/pgf/number format/every relative/.style=std}
```

Note that rounding is turned off when the resulting style is being evaluated (since `relative` already rounded the number).

Although supported, I discourage from using `fixed zerofill` or `sci zerofill` in this context – it may lead to a suggestion of higher precision than is actually used (because `fixed zerofill` might simply add `.00` although there was a different information before `relative` rounded the result).

`/pgf/number format/relative style={`⟨*options*⟩`}` (no default)

The same as `every relative/.append style={`⟨*options*⟩`}`.

`/pgf/number format/fixed relative` (no value)

Configures `\pgfmathprintnumber` to format numbers in a similar way to the `fixed` style, but the `precision` is interpreted relatively to the number's exponent.

The motivation is to get the same rounding effect as for `sci`, but to display the number in the `fixed` style:

1,000 100 0.00001 0.0101 1.24 1,000 1,010

```
\pgfkeys{/pgf/number format/.cd,fixed relative,precision=3}
\pgfmathprintnumber{1000.0123}\hspace{1em}
\pgfmathprintnumber{100.0567}\hspace{1em}
\pgfmathprintnumber{0.000010003452}\hspace{1em}
\pgfmathprintnumber{0.010073452}\hspace{1em}
\pgfmathprintnumber{1.23567}\hspace{1em}
\pgfmathprintnumber{1003.75}\hspace{1em}
\pgfmathprintnumber{1006.75}\hspace{1em}
```

The effect of `fixed relative` is that the number is rounded to *exactly* the first ⟨*precision*⟩ non-zero digits, no matter how many leading zeros the number might have.

Use `fixed relative` if you want `fixed` and if you know that only the first n digits are correct. Use `sci` if you need a scientific display style and only the first n digits are correct.

Note that `fixed relative` ignores the `fixed zerofill` flag.

See also the `relative*` key. Note that the `relative={`⟨*exponent*⟩`}` key explicitly moves the period to some designated position before it attempts to round the number. Afterwards, it "rounds from the right", i.e. it rounds to that explicitly chosen digit position. In contrast to that, `fixed relative` "rounds from the left": it takes the *first* non-zero digit, temporarily places the period after this digit, and rounds that number. The rounding style `fixed` leaves the period where it is, and rounds everything behind that digit. The `sci` style is similar to `fixed relative`.

`/pgf/number format/int detect` (no value)

Configures `\pgfmathprintnumber` to detect integers automatically. If the input number is an integer, no period is displayed at all. If not, the scientific format is chosen.

15 20 $2.04 \cdot 10^{1}$ $1 \cdot 10^{-2}$ 0

```
\pgfkeys{/pgf/number format/.cd,int detect,precision=2}
\pgfmathprintnumber{15}\hspace{1em}
\pgfmathprintnumber{20}\hspace{1em}
\pgfmathprintnumber{20.4}\hspace{1em}
\pgfmathprintnumber{0.01}\hspace{1em}
\pgfmathprintnumber{0}
```

`\pgfmathifisint{`⟨*number constant*⟩`}{`⟨*true code*⟩`}{`⟨*false code*⟩`}`

A command which does the same check as `int detect`, but it invokes ⟨*true code*⟩ if the ⟨*number constant*⟩ actually is an integer and the ⟨*false code*⟩ if not.

As a side-effect, `\pgfretval` will contain the parsed number, either in integer format or as parsed floating point number.

The argument ⟨*number constant*⟩ will be parsed with `\pgfmathfloatparsenumber`.

15 is an int: 15. 15.5 is no int

```
15 \pgfmathifisint{15}{is an int: \pgfretval.}{is no int}\hspace{1em}
15.5 \pgfmathifisint{15.5}{is an int: \pgfretval.}{is no int}
```

/pgf/number format/int trunc (no value)

Truncates every number to integers (discards any digit after the period).

4 0 0 24,415 123,456

```
\pgfkeys{/pgf/number format/.cd,int trunc}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

/pgf/number format/frac (no value)

Displays numbers as fractionals.

$\frac{1}{3}$ $\frac{1}{2}$ $\frac{16}{75}$ $\frac{3}{25}$ $\frac{2}{75}$ $-\frac{1}{75}$ $\frac{18}{25}$ $\frac{1}{15}$ $\frac{2}{15}$ $-\frac{1}{75}$ $3\frac{1}{3}$ $1\frac{22657}{96620}$ 1 -6

```
\usetikzlibrary {fpu}
\pgfkeys{/pgf/number format/frac}
\pgfmathprintnumber{0.333333333333333}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{2.133333333333325e-01}\hspace{1em}
\pgfmathprintnumber{0.12}\hspace{1em}
\pgfmathprintnumber{2.666666666666646e-02}\hspace{1em}
\pgfmathprintnumber{-1.333333333333334e-02}\hspace{1em}
\pgfmathprintnumber{7.200000000000000e-01}\hspace{1em}
\pgfmathprintnumber{6.666666666666667e-02}\hspace{1em}
\pgfmathprintnumber{1.333333333333333e-01}\hspace{1em}
\pgfmathprintnumber{-1.333333333333333e-02}\hspace{1em}
\pgfmathprintnumber{3.3333333}\hspace{1em}
\pgfmathprintnumber{1.2345}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{-6}
```

/pgf/number format/frac TeX={⟨*macro*⟩} (no default, initially `\frac`)

Allows to use a different implementation for `\frac` inside of the `frac` display type.

/pgf/number format/frac denom=⟨*int*⟩ (no default, initially `empty`)

Allows to provide a custom denominator for `frac`.

$\frac{1}{10}$ $\frac{5}{10}$ $1\frac{2}{10}$ $-\frac{6}{10}$ $-1\frac{4}{10}$

```
\usetikzlibrary {fpu}
\pgfkeys{/pgf/number format/.cd,frac, frac denom=10}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-0.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

/pgf/number format/frac whole=true|false (no default, initially `true`)

Configures whether complete integer parts shall be placed in front of the fractional part. In this case, the fractional part will be less then 1. Use `frac whole=false` to avoid whole number parts.

$\frac{201}{10}$ $\frac{11}{2}$ $\frac{6}{5}$ $-\frac{28}{5}$ $-\frac{7}{5}$

```
\usetikzlibrary {fpu}
\pgfkeys{/pgf/number format/.cd,frac, frac whole=false}
\pgfmathprintnumber{20.1}\hspace{1em}
\pgfmathprintnumber{5.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-5.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

`/pgf/number format/frac shift`=`{`⟨*integer*⟩`}` (no default, initially `4`)

In case you experience problems because of stability problems, try experimenting with a different `frac shift`. Higher shift values k yield higher sensitivity to inaccurate data or inaccurate arithmetic.

Technically, the following happens. If $r < 1$ is the fractional part of the mantissa, then a scale $i = 1/r \cdot 10^k$ is computed where k is the shift; fractional parts of i are neglected. The value $1/r$ is computed internally, its error is amplified.

If you still experience stability problems, use `\usepackage{fp}` in your preamble. The `frac` style will then automatically employ the higher absolute precision of `fp` for the computation of $1/r$.

`/pgf/number format/precision`=`{`⟨*number*⟩`}` (no default)

Sets the desired rounding precision for any display operation. For scientific format, this affects the mantissa.

`/pgf/number format/sci precision`=⟨*number or empty*⟩ (no default, initially `empty`)

Sets the desired rounding precision only for `sci` styles.

Use `sci precision={}` to restore the initial configuration (which uses the argument provided to `precision` for all number styles).

`/pgf/number format/read comma as period`=`true`|`false` (no default, initially `false`)

This is one of the few keys which allows to customize the number parser. If this switch is turned on, a comma is read just as a period.

1,234.56

```
\pgfkeys{/pgf/number format/read comma as period}
\pgfmathprintnumber{1234,56}
```

This is typically undesired as it can cause side-effects with math parsing instructions. However, it is supported to format input numbers or input tables. Consider `use comma` to typeset the result with a comma as well.

1.234,56

```
\pgfkeys{/pgf/number format/.cd,
    read comma as period,
    use comma}
\pgfmathprintnumber{1234,56}
```

97.1 Changing display styles

You can change the way how numbers are displayed. For example, if you use the '`fixed`' style, the input number is rounded to the desired precision and the current fixed point display style is used to typeset the number. The same is applied to any other format: first, rounding routines are used to get the correct digits, afterwards a display style generates proper TEX-code.

`/pgf/number format/set decimal separator`=`{`⟨*text*⟩`}` (no default)

Assigns `{`⟨*text*⟩`}` as decimal separator for any fixed point numbers (including the mantissa in sci format).

Use `\pgfkeysgetvalue{/pgf/number format/set decimal separator}\value` to get the current separator into `\value`.

`/pgf/number format/dec sep`=`{`⟨*text*⟩`}` (style, no default)

Just another name for `set decimal separator`.

`/pgf/number format/set thousands separator={`⟨*text*⟩`}` (no default)

Assigns `{`⟨*text*⟩`}` as thousands separator for any fixed point numbers (including the mantissa in sci format).

1234.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={}}
\pgfmathprintnumber{1234.56}
```

1234567890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={}}
\pgfmathprintnumber{1234567890}
```

1.234.567.890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={.}}
\pgfmathprintnumber{1234567890}
```

1, 234, 567, 890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={,}}
\pgfmathprintnumber{1234567890}
```

1,234,567,890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={{{,}}}}
\pgfmathprintnumber{1234567890}
```

The last example employs commas and disables the default comma-spacing.

Use `\pgfkeysgetvalue{/pgf/number format/set thousands separator}\value` to get the current separator into `\value`.

`/pgf/number format/1000 sep={`⟨*text*⟩`}` (style, no default)

Just another name for `set thousands separator`.

`/pgf/number format/1000 sep in fractionals={`⟨*boolean*⟩`}` (no default, initially `false`)

Configures whether the fractional part should also be grouped into groups of three digits.

The value `true` will active the `1000 sep` for both, integer and fractional parts. The value `false` will active `1000 sep` only for the integer part.

1 234.123 456 7

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    precision=999,
    set thousands separator={\,},
    1000 sep in fractionals,
    }
\pgfmathprintnumber{1234.1234567}
```

1 234.123 456 700

```
\pgfkeys{/pgf/number format/.cd,
    fixed,fixed zerofill,
    precision=9,
    set thousands separator={\,},
    1000 sep in fractionals,
    }
\pgfmathprintnumber{1234.1234567}
```

/pgf/number format/min exponent for 1000 sep={⟨*number*⟩} (no default, initially 0)

Defines the smallest exponent in scientific notation which is required to draw thousand separators. The exponent is the number of digits minus one, so ⟨*number*⟩ = 4 will use thousand separators starting with $1e4 = 10000$.

5 000; 1 000 000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=0}
\pgfmathprintnumber{5000}; \pgfmathprintnumber{1000000}
```

1000; 5000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=4}
\pgfmathprintnumber{1000}; \pgfmathprintnumber{5000}
```

10 000; 1 000 000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=4}
\pgfmathprintnumber{10000}; \pgfmathprintnumber{1000000}
```

A value of 0 disables this feature (negative values are ignored).

/pgf/number format/use period (no value)

A predefined style which installs periods "." as decimal separators and commas "," as thousands separators. This style is the default.

12.35

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period}
\pgfmathprintnumber{12.3456}
```

1,234.56

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period}
\pgfmathprintnumber{1234.56}
```

/pgf/number format/use comma (no value)

A predefined style which installs commas "," as decimal separators and periods "." as thousands separators.

12,35

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma}
\pgfmathprintnumber{12.3456}
```

1.234,56

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma}
\pgfmathprintnumber{1234.56}
```

/pgf/number format/skip 0.={⟨*boolean*⟩} (no default, initially `false`)

Configures whether numbers like 0.1 shall be typeset as .1 or not.

.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,precision=2,
    skip 0.}
\pgfmathprintnumber{0.56}
```

0.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,precision=2,
    skip 0.=false}
\pgfmathprintnumber{0.56}
```

/pgf/number format/showpos={⟨*boolean*⟩} (no default, initially `false`)

Enables or disables the display of plus signs for non-negative numbers.

$+12.35$

```
\pgfkeys{/pgf/number format/showpos}
\pgfmathprintnumber{12.345}
```

12.35

```
\pgfkeys{/pgf/number format/showpos=false}
\pgfmathprintnumber{12.345}
```

$+1.23 \cdot 10^{1}$

```
\pgfkeys{/pgf/number format/.cd,showpos,sci}
\pgfmathprintnumber{12.345}
```

/pgf/number format/print sign={⟨*boolean*⟩} (style, no default)

A style which is simply an alias for `showpos={`⟨*boolean*⟩`}`.

/pgf/number format/sci 10e (no value)

Uses $m \cdot 10^{e}$ for any number displayed in scientific format.

$1.23 \cdot 10^{1}$

```
\pgfkeys{/pgf/number format/.cd,sci,sci 10e}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci 10^e (no value)

The same as '`sci 10e`'.

/pgf/number format/sci e (no value)

Uses the '$1e{+}0$' format which is generated by common scientific tools for any number displayed in scientific format.

$1.23e{+}1$

```
\pgfkeys{/pgf/number format/.cd,sci,sci e}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci E (no value)

The same with an uppercase '`E`'.

$1.23E{+}1$

```
\pgfkeys{/pgf/number format/.cd,sci,sci E}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci subscript (no value)

Typesets the exponent as subscript for any number displayed in scientific format. This style requires very little space.

1.23_{1}

```
\pgfkeys{/pgf/number format/.cd,sci,sci subscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci superscript (no value)

Typesets the exponent as superscript for any number displayed in scientific format. This style requires very little space.

1.23^{1}

```
\pgfkeys{/pgf/number format/.cd,sci,sci superscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci generic={⟨*keys*⟩} (no default)

Allows to define an own number style for the scientific format. Here, ⟨*keys*⟩ can be one of the following choices (omit the long key prefix):

/pgf/number format/sci generic/mantissa sep={⟨*text*⟩} (no default, initially `empty`)

Provides the separator between a mantissa and the exponent. It might be `\cdot`, for example,

/pgf/number format/sci generic/exponent={⟨*text*⟩} (no default, initially `empty`)

Provides text to format the exponent. The actual exponent is available as argument `#1` (see below).

$1.23 \times 10^{1};1.23 \times 10^{-4}$

```
\pgfkeys{
    /pgf/number format/.cd,
    sci,
    sci generic={mantissa sep=\times,exponent={10^{#1}}}}
\pgfmathprintnumber{12.345};
\pgfmathprintnumber{0.00012345}
```

The ⟨*keys*⟩ can depend on three parameters, namely on `#1` which is the exponent, `#2` containing the flags entity of the floating point number and `#3` is the (unprocessed and unformatted) mantissa.

Note that `sci generic` is *not* suitable to modify the appearance of fixed point numbers, nor can it be used to format the mantissa (which is typeset like fixed point numbers). Use `dec sep`, `1000 sep` and `print sign` to customize the mantissa.

/pgf/number format/retain unit mantissa=true|false (no default, initially `true`)

Allows to omit a unit mantissa.

$1.05 \cdot 10^{1};10^{1};1.01 \cdot 10^{3};-10^{3};$

```
\pgfkeys{
    /pgf/number format/.cd,
    sci, retain unit mantissa=false}
\pgfmathprintnumber{10.5};
\pgfmathprintnumber{10};
\pgfmathprintnumber{1010};
\pgfmathprintnumber[precision=1]{-1010};
```

The feature is applied after rounding to the desired precision: if the remaining mantissa is equal to 1, it will be omitted. It applies to all styles involving the scientific format (including `std`).

/pgf/number format/@dec sep mark={⟨*text*⟩} (no default)

Will be placed right before the place where a decimal separator belongs to. However, {⟨*text*⟩} will be inserted even if there is no decimal separator. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the decimal separator! Use `dec sep` instead.

/pgf/number format/@sci exponent mark={⟨*text*⟩} (no default)

Will be placed right before exponents in scientific notation. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the exponent!

/pgf/number format/assume math mode={⟨*boolean*⟩} (default `true`)

Set this to `true` if you don't want any checks for math mode. The initial setting checks whether math mode is active using `\pgfutilensuremath` for each final number.

Use `assume math mode=true` if you know that math mode is active. In that case, the final number is typeset as-is, no further checking is performed.

/pgf/number format/verbatim (style, no value)

A style which configures the number printer to produce verbatim text output, i.e., it doesn't contain TEX macros.

1.23e1;1.23e-4;3.27e6

```
\usetikzlibrary {fpu}
\pgfkeys{
    /pgf/fpu,
    /pgf/number format/.cd,
    sci,
    verbatim}
\pgfmathprintnumber{12.345};
\pgfmathprintnumber{0.00012345};
\pgfmathparse{exp(15)}
\pgfmathprintnumber{\pgfmathresult}
```

The style resets `1000 sep`, `dec sep`, `print sign`, `skip 0.` and sets `assume math mode`. Furthermore, it installs a `sci generic` format for verbatim output of scientific numbers.

However, it will still respect `precision`, `fixed zerofill`, `sci zerofill` and the overall styles `fixed`, `sci`, `int detect` (and their variants). It might be useful if you intend to write output files.

98 Object-Oriented Programming

This section describes the `oo` module.

```
\usepgfmodule{oo}  % LaTeX and plain TeX when using pure pgf
\usepgfmodule[oo]  % ConTeXt when using pure pgf
```

This module defines a relatively small set of TEX commands for defining classes, methods, attributes and objects in the sense of object-oriented programming.

In this chapter it is assumed that you are familiar with the basics of a typical object-oriented programming language like Java, C++ or Eiffel.

98.1 Overview

TEX does not support object-oriented programming, presumably because it was written at a time when this style of programming was not yet "en vogue". When one is used to the object-oriented style of thinking, some programming constructs in TEX often seem overly complicated. The object-oriented programming module of PGF may help here. It is written completely using simple TEX macros and is, thus, perfectly portable. This also means, however, that it is not particularly fast (but not too slow either), so you should use it only for non-time-critical things.

Basically, the oo-system supports *classes* (in the object-oriented sense, this has nothing to do with LATEX-classes), *methods*, *constructors*, *attributes*, *objects*, *object identities*, and (thanks to Sašo Živanović) *inheritance* and *overloading*.

The first step is to define a class, using the macro `\pgfooclass` (all normal macros in PGF's object-oriented system start with `\pgfoo`). This macro gets the name of a class and in its body a number of *methods* are defined. These are defined using the `\method` macro (which is defined only inside such a class definition) and they look a bit like method definitions in, say, Java. Object attributes are declared using the `\attribute` command, which is also defined only inside a class definition.

Once a class has been defined, you can create objects of this class. Objects are created using `\pgfoonew`. Such an object has many characteristics of objects in a normal object-oriented programming language: Each object has a *unique identity*, so when you create another object, this object is completely distinct from all other objects. Each object also has a set of private attributes, which may change over time. Suppose, for instance, that we have a `point` class. Then creating a new object (called an instance) of this class would typically have an `x`-attribute and a `y`-attribute. These can be changed over time. Creating another instance of the `point` class creates another object with its own `x`- and `y`-attributes.

Given an object, you can call a method for this object. Inside the method the attributes of the object for which the method is being called can be accessed.

The life of an object always ends with the end of the TEX scope in which it was created. However, changes to attribute values are not local to scopes, so when you change an attribute anywhere, this change persists till the end of the life of the object or until the attribute is changed again.

98.2 A Running Example: The Stamp Class

As a running example we will develop a `stamp` class and `stamp` objects. The idea is that a stamp object is able to "stamp something" on a picture. This means that a stamp object has an attribute storing the "stamp text" and there is a method that asks the object to place this text somewhere on a canvas. The method can be called repeatedly and there can be several different stamp objects, each producing a different text. Stamp objects can either be created dynamically when needed or a library might define many such objects in an outer scope.

Such stamps are similar to many things present in PGF such as arrow tips, patterns, or shadings and, indeed, these could all have been implemented in this object-oriented fashion (which might have been better, but the object-oriented subsystem is a fairly new addition to PGF).

98.3 Classes

We start with the definition of the `stamp` class. This is done using the `\pgfooclass` macro:

`\pgfooclass(`⟨*list of superclasses*⟩`){`⟨*class name*⟩`}{`⟨*body*⟩`}`

This command defines a class named ⟨*class name*⟩. The name of the class can contain spaces and most other characters, but no periods. So, valid class names are `MyClass` or `my class` or `Class_C++_emulation??1`. The ⟨*list of superclasses*⟩ is optional just like the parenthesis around it.

The ⟨*body*⟩ is actually just executed, so any normal TEX-code is permissible here. However, while the ⟨*body*⟩ is being executed, the macros `\method` and `\attribute` are set up so that they can be used to define methods and attributes for this class (the original meanings are restored afterward).

The definition of a class is local to the scope where the class has been defined.

```
\pgfooclass{stamp}{
  % This is the class stamp

  \attribute text;
  \attribute rotation angle=20;

  \method stamp(#1) { % The constructor
    ...
  }

  \method apply(#1,#2) { % Causes the stamp to be shown at coordinate (#1,#2)
    ...
  }
}

% We can now create objects of type "stamp"
```

Concerning the list of base classes, the Method Resolution Order (MRO) is computed using the C3 algorithm also used in Python, v2.3 and higher. The linearization computed by the algorithm respects both local precedence ordering and monotonicity. Resolution of both methods and attributes depends on the MRO: when a method method name is called on an object of class C, the system invokes method method name from the first class in the MRO of C which defines method method name; when an object is created, each attribute `attr` is initialized to the value specified in the first class in the MRO of C which declares attribute `attr`.

The ⟨*body*⟩ of a class usually just consists of calls to the macros `\attribute` and `\method`, which will be discussed in more detail in later sections.

98.4 Objects

Once a class has been declared, we can start creating objects for this class. For this the `\pgfoonew` command can be used, which has a peculiar syntax:

`\pgfoonew`⟨*object handle or attribute*⟩`=new` ⟨*class name*⟩`(`⟨*constructor arguments*⟩`)`

Causes a new object to be created. The class of the object will be ⟨*class name*⟩, which must previously have been declared using `\pgfooclass`. Once the object has been created, the constructor method of the object will be called with the parameter list set to ⟨*constructor arguments*⟩.

The resulting object is stored internally and its lifetime will end exactly at the end of the current scope.

Here is an example in which three stamp objects are created.

```
\pgfoonew \firststamp=new stamp()
\pgfoonew \secondstamp=new stamp()
{
  \pgfoonew \thirdstamp=new stamp()
  ...
}
% \thirdstamp no longer exists, but \firststamp and \secondstamp do
% even if you try to store \thirdstamp in a global variable, trying
% to access it will result in an error.
```

The optional ⟨*object handle or attribute*⟩ can either be an ⟨*object handle*⟩ or an ⟨*attribute*⟩. When an ⟨*object handle*⟩ is given, it must be a normal TEX macro name that will "point" to the object (handles are discussed in more detail in Section 98.7). You can use this macro to call methods of the object as discussed in the following section. When an ⟨*attribute*⟩ is given, it must be given in curly braces (the curly braces are used to detect the presence of an attribute). In this case, a handle to the newly created object is stored in this attribute.

```
\pgfooclass{foo}
{
  \attribute stamp obj;
  \attribute another object;

  \method foo() {
    \pgfoonew{stamp obj}=new stamp()
    \pgfoonew{another object}=new bar()
  }
  ...
}
```

\pgfoogc

This command causes the "garbage collector" to be invoked. The job of this garbage collector is to free the global TEX-macros that are used by "dead" objects (objects whose life-time has ended). This macro is called automatically after every scope in which an object has been created, so you normally do not need to call this macro yourself.

98.5 Methods

Methods are defined inside the body of classes using the following command:

\method⟨*method name*⟩(⟨*parameter list*⟩){⟨*method body*⟩}

This macro, which is only defined inside a class definition, defines a new method named ⟨*method name*⟩. Just like class names, method names can contain spaces and other characters, so ⟨*method names*⟩ like `put_stamp_here` or `put stamp here` are both legal.

Three method names are special: First, a method having either the same name as the class or having the name `init` is called the *constructor* of the class. There are (currently) no destructors; objects simply become "undefined" at the end of the scope in which they have been created. The other two methods are called `get id` and `get handle`, which are always automatically defined and which you cannot redefine. They are discussed in Section 98.7.

Overloading of methods by differing numbers of parameters is not possible, that is, it is illegal to have two methods inside a single class with the same name (despite possibly different parameter lists). However, two different classes may contain a method with the same name, that is, classes form namespaces for methods. Also, a class can (re)implement a method from a superclass.

The ⟨*method name*⟩ must be followed by a ⟨*parameter list*⟩ in parentheses, which must be present even when the ⟨*parameter list*⟩ is empty. The ⟨*parameter list*⟩ is actually a normal TEX parameter list that will be matched against the parameters inside the parentheses upon method invocation and, thus, could be something like `#1#2 foo #3 bar.`, but a list like `#1,#2,#3` is more customary. By setting the parameter list to just `#1` and then calling, say, `\pgfkeys{#1}` at the beginning of a method, you can implement Objective-C-like named parameters.

When a method is called, the ⟨*body*⟩ of the method will be executed. The main difference to a normal macro is that while the ⟨*body*⟩ is executed, a special macro called `\pgfoothis` is set up in such a way that it references the object for which the method is executed.

In order to call a method for an object, you first need to create the object and you need a handle for this object. In order to invoke a method for this object, a special syntax is used that is similar to Java or C++ syntax:

⟨*object handle*⟩.⟨*super class*⟩.⟨*method name*⟩(⟨*parameters*⟩)

This causes the method ⟨*method name*⟩ to be called for the object referenced by the ⟨*object handle*⟩. The method is the one defined in the class of the object or, if it is not defined there, the method defined in the superclasses of the object's class (if there are several superclasses that define the same method, the method resolution order is used to determine which one gets called). If the optional ⟨*super class*⟩ is specified, the method implementation of that class will be used rather than the implementation in the object's class. The ⟨*parameters*⟩ are matched against the parameters of the method and, then, the method body is executed. The execution of the method body is *not* done inside a scope, so the effects of a method body persist.

```
\pgfooclass{stamp}{
  % This is the class stamp

  \method stamp() { % The constructor
  }

  \method apply(#1,#2) { % Causes the stamp to be shown at coordinate (#1,#2)
    % Draw the stamp:
    \node [rotate=20,font=\huge] at (#1,#2) {Passed};
  }
}

\pgfoonew \mystamp=new stamp()

\begin{tikzpicture}
  \mystamp.apply(1,2)
  \mystamp.apply(3,4)
\end{tikzpicture}
```

Inside a method, you can call other methods. If you have a handle for another object, you can simply call it in the manner described above. In order to call a method of the current object, you can use the special object handle `\pgfoothis`.

`\pgfoothis`

This object handle is well-defined only when a method is being executed. There, it is then set to point to the object for which the method is being called, which allows you to call another method for the same object.

```
\pgfooclass{stamp}{
  % This is the class stamp

  \method stamp() {}

  \method apply(#1,#2) {
    \pgfoothis.shift origin(#1,#2)

    % Draw the stamp:
    \node [rotate=20,font=\huge] {Passed};
  }

  % Private method:
  \method shift origin(#1,#2) {
    \tikzset{xshift=#1,yshift=#2}
  }
}
```

`\pgfoosuper(`⟨*class*⟩`,`⟨*object handle*⟩`).`⟨*method name*⟩`(`⟨*arguments*⟩`)`

This macro gives you finer control over which method gets invoked in case of multiple inheritance. This macro calls ⟨*method name*⟩ of the object specified by ⟨*object handle*⟩, but which implementation of the method is called is determined as follows: it will be the implementation in the first class (in the method resolution order) after ⟨*class*⟩ that defines ⟨*method name*⟩.

98.6 Attributes

Every object has a set of attributes, which may change over time. Attributes are declared using the `\attribute` command, which, like the `\method` command, is defined only inside the scope of `\pgfooclass`. Attributes can be modified (only) by methods. To take the `stamp` example, an attribute of a `stamp` object might be the text that should be stamped when the `apply` method is called.

When an attribute is changed, this change is *not* local to the current TEX group. Changes will persist till the end of the object's life or until the attribute is changed once more.

To declare an attribute you should use the `\attribute` command:

`\attribute`⟨*attribute name*⟩`=`⟨*initial value*⟩`;`

This command can only be given inside the body of an `\pgfooclass` command. It declares the attribute named ⟨*attribute name*⟩. This name, like method or class names, can be quite arbitrary, but should not contain periods. Valid names are `an_attribute?` or `my attribute`.

You can optionally specify an ⟨*initial value*⟩ for the attribute; if none is given, the empty string is used automatically. The initial value is the value that the attribute will have just after the object has been created and before the constructor is called.

```
\pgfooclass{stamp}{
  % This is the class stamp

  \attribute text;
  \attribute rotation angle = 20;

  \method stamp(#1) {
    \pgfooset{text}{#1} % Set the text
  }

  \method apply(#1,#2) {
    \pgfoothis.shift origin(#1,#2)

    % Draw the stamp:
    \node [rotate=\pgfoovalueof{rotation angle},font=\huge]
      {\pgfoovalueof{text}};
  }

  \method shift origin(#1,#2) { ... }

  \method set rotation (#1) {
    \pgfooset{rotation angle}{#1}
  }
}
```

Attributes can be set and read only inside methods, it is not possible to do so using an object handle. Spoken in terms of traditional object-oriented programming, attributes are always private. You need to define getter and setter methods if you wish to read or modify attributes.

Reading and writing attributes is not done using the "dot-notation" that is used for method calls. This is mostly due to efficiency reasons. Instead, a set of special macros is used, all of which can *only be used inside methods*.

`\pgfooset{`⟨*attribute*⟩`}{`⟨*value*⟩`}`

Sets the ⟨*attribute*⟩ of the current object to ⟨*value*⟩.

```
\method set rotation (#1) {
  \pgfooset{rotation angle}{#1}
}
```

`\pgfooeset{`⟨*attribute*⟩`}{`⟨*value*⟩`}`

Performs the same action as `\pgfooset` but in an `\edef` full expansion context.

`\pgfooappend{`⟨*attribute*⟩`}{`⟨*value*⟩`}`

This method adds the given ⟨*value*⟩ to the ⟨*attribute*⟩ at the end.

`\pgfooprefix{`⟨*attribute*⟩`}{`⟨*value*⟩`}`

This method adds the given ⟨*value*⟩ to the ⟨*attribute*⟩ at the beginning.

`\pgfoolet{`⟨*attribute*⟩`}{`⟨*macro*⟩`}`

Sets the ⟨*attribute*⟩ of the current value to the current value of ⟨*macro*⟩ using TEX's `\let` command.

```
\method foo () {
  \pgfoolet{my func}\myfunc
  % Changing \myfunc now has no effect on the value of attribute my func
}
```

`\pgfoovalueof{`⟨*attribute*⟩`}`

Expands (eventually) to the current value of ⟨*attribute*⟩ of the current object.

```
\method apply(#1,#2) {
  \pgfoothis.shift origin(#1,#2)

  \node [rotate=\pgfoovalueof{rotation angle},font=\huge]
    {\pgfoovalueof{text}};
}
```

\pgfooget{⟨*attribute*⟩}{⟨*macro*⟩}

Reads the current value of ⟨*attribute*⟩ and stores the result in ⟨*macro*⟩.

```
...
  \method get rotation (#1) {
    \pgfooget{rotation angle}{#1}
  }
...

\mystamp.get rotation(\therotation)
``\therotation'' is now ``20'' (or whatever).
```

98.7 Identities

Every object has a unique identity, which is simply an integer. It is possible to retrieve the object id using the `get id` method (discussed below), but normally you will not need to do so because the id itself cannot be used to access an object. Rather, you access objects via their methods and these, in turn, can only be called via object handles.

Object handles can be created in four ways:

1. Calling \pgfoonew⟨*object handle*⟩=... will cause ⟨*object handle*⟩ to be a handle to the newly created object.
2. Using \let to create an alias of an existing object handle: If \mystamp is a handle, saying \let\myotherstamp=\mystamp creates a second handle to the same object.
3. \pgfooobj{⟨*id*⟩} can be used as an object handle to the object with the given ⟨*id*⟩.
4. Using the `get handle` method to create a handle to a given object.

Let us have a look at the last two methods.

\pgfooobj{⟨*id*⟩}

Provided that ⟨*id*⟩ is the id of an existing object (an object whose life-time has not expired), calling this command yields a handle to this object. The handle can then be used to call methods:

```
% Create a new object:
\pgfoonew \mystamp=new stamp()

% Get the object's id and store it in \myid:
\mystamp.get id(\myid)

% The following two calls have the same effect:
\mystamp.apply(1,1)
\pgfooobj{\myid}.apply(1,1)
```

The `get id` method can be used to retrieve the id of an object. This method is predefined for every class and you should not try to define a method of this name yourself.

Method `get id`(⟨*macro*⟩) (predefined for all classes)

Calling ⟨*obj*⟩`.get id(`⟨*macro*⟩`)` stores the id ⟨*obj*⟩ in ⟨*macro*⟩. This is mainly useful when you wish to store an object for a longer time and you cannot guarantee that any handle that you happen to have for this object will be available later on.

The only way to use the retrieved id later on is to call \pgfooobj.

Different object that are alive (that are still within the scope in which they were created) will always have different ids, so you can use the id to test for equality of objects. However, after an object has been destroyed because its scope has ended, the same id may be used again for newly created objects.

Here is a typical application where you need to call this method: You wish to collect a list of objects for which you wish to call a specific method from time to time. For the collection process you wish to offer a macro called `\addtoobjectlist`, which takes an object handle as parameter. It is quite easy to store this handle somewhere, but a handle is, well, just a handle. Typically, shortly after the call to `\addtoobjectlist` the handle will no longer be valid or even exist, even though the object still exists. In this case, you wish to store the object id somewhere instead of the handle. Thus, for the object passed to `\addtoobjectlist` you call the `get id` method and store the resulting id, rather than the handle.

There is a second predefined method, called `get handle`, which is also used to create object handles.

Method `get handle({`⟨*macro name*⟩`})` (predefined for all classes)

Calling this method for an object will cause ⟨*macro name*⟩ to become a handle to the given object. For any object handle `\obj` – other than `\pgfoothis` – the following two have the same effect:

1. `\let`⟨*macro name*⟩`=\obj`
2. `\obj.get handle(`⟨*macro name*⟩`)`

The first method is simpler and faster. However, for `\pgfoothis` there is a difference: The call `\pgfoothis.get handle(`⟨*macro name*⟩`)` will cause ⟨*macro name*⟩ to be an object handle to the current object and will persist to be so even after the method is done. By comparison, `\let`⟨*macro name*⟩`=\pgfoothis` causes `\obj` to be the same as the very special macro `\pgfoothis`, so `\obj` will always refer to the current object, which may change over time.

98.8 The Object Class

The object-oriented module predefines a basic class `object` that can be used as a base class in different context.

Class `object`

This class current only implements one method:

Method `copy(`⟨*handle*⟩`)`

Creates a new object and initializes the values of its (declared) attributes to the values of the original. The method takes one argument: a control sequence which receives the handle of the copy.

98.9 The Signal Class

In addition to the basic mechanism for defining and using classes and object, the class `signal` is predefined. It implements a so-called signal–slot mechanism.

Class `signal`

This class is used to implement a simple signal–slot mechanism. The idea is the following: From time to time special things happen about which a number of objects need to be informed. Different things can happen and different object will be interested in these things. A `signal` object can be used to signal that such special things of a certain kind have happened. For example, one signal object might be used to signal the event that "a page has been shipped out". Another signal might be used to signal that "a figure is about to be typeset", and so on.

Objects can "tune in" to signals. They do so by *connecting* one of their methods (then called a *slot*) to the signal. Then, whenever the signal is *emitted*, the method of the connected object(s) get called. Different objects can connect different slots to the same signal as long as the argument lists will fit. For example, the object that is used to signal the "end of page has been reached" might emit signals that have, say, the box number in which the finished page can be found as a parameter (actually, the finished page is always in box 255). Then one object could connect a method `handle page(#1)` to this signal, another might connect the method `emergency action(#1)` to this signal, and so on.

Currently, it is not possible to "unregister" or "detach" a slot from a signal, that is, once an object has been connect to a signal, it will continue to receive emissions of this signal till the end of the life-time of the signal. This is even true when the object no longer exists (but the signal does), so care must be taken that signal objects are always created after the objects that are listening to them.

Constructor `signal()`

The constructor does nothing.

Method `connect(`⟨*object handle*⟩`,`⟨*method name*⟩`)`

This method gets an ⟨*object handle*⟩ as parameter and a ⟨*method name*⟩ of this object. It will queue the object-method pair in an internal list and each time the signal emits something, this object's method is called.

Be careful not to pass `\pgfoothis` as ⟨*object handle*⟩. This would cause the signal object to connect to itself. Rather, if you wish to connect a signal to a method of the current object you first need to create an alias using the `get handle` method:

```
\pgfooclass{some class}{
  \method some class() {
    \pgfoothis.get handle(\me)
    \somesignal.connect(\me,foo)
    \anothersignal.connect(\me,bar)
  }
  \method foo () {}
  \method bar (#1,#2) {}
}
\pgfoonew \objA=new some class()
\pgfoonew \objB=new some class()
```

Method `emit(`⟨*arguments*⟩`)`

This method emits a signal to all connected slots. This means that for all objects that have previously been connected via a call of `connect`, the method (slot) that was specified during the call of `connect` is invoked with given ⟨*arguments*⟩.

```
\anothersignal.emit(1,2)
% will call \objA.bar(1,2) and \objB.bar(1,2)
```

98.10 Implementation Notes

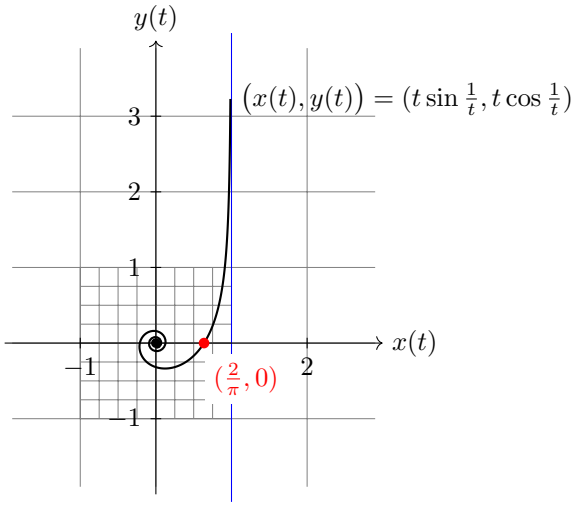
For the curious, here are some notes on how the oo-system is implemented:

- There is an object id counter that gets incremented each time an object is created. However, this counter is local to the current scope, which means that it is reset at the end of each scope, corresponding to the fact that at the end of a scope all objects created in this scope become invalid. Newly created objects will then have the same id as "deleted" objects.
- Attributes are stored globally. For each attribute of each object there is a macro whose name is composed of the object's id and the attribute name. Changes to object attributes are always global.
- A call to the garbage collector causes a loop to be executed that tries to find objects whose object number is larger than the current maximum alive objects. The global attributes of these objects are then freed (set to `\relax`) by calling a special internal method of these (dead) objects.

  The garbage collector is automatically called after each group in which an object was created using `\aftergroup`.
- When a method is called, before the method call some code is executed that sets a global counter storing the current object id to the object id of the object being called. After the method call some code is inserted that restores the global counter to its original value. This is done without scopes, so some tricky `\expandafter` magic is needed. Note that, because of this process, you cannot use commands like `\pgfutil@ifnextchar` at the end of a method.
- An object handle contains just the code to set up and restore the current object number to the number of the object being called.

Part IX
The Basic Layer

by Till Tantau




```
\begin{tikzpicture}
  \draw[gray,very thin] (-1.9,-1.9) grid (2.9,3.9)
          [step=0.25cm] (-1,-1) grid (1,1);
  \draw[blue] (1,-2.1) -- (1,4.1); % asymptote

  \draw[->] (-2,0) -- (3,0) node[right] {$x(t)$};
  \draw[->] (0,-2) -- (0,4) node[above] {$y(t)$};

  \foreach \pos in {-1,2}
    \draw[shift={(\pos,0)}] (0pt,2pt) -- (0pt,-2pt) node[below] {$\pos$};

  \foreach \pos in {-1,1,2,3}
    \draw[shift={(0,\pos)}] (2pt,0pt) -- (-2pt,0pt) node[left] {$\pos$};

  \fill (0,0) circle [radius=0.064cm];
  \draw[thick,parametric,domain=0.4:1.5,samples=200]
    % The plot is reparameterised such that there are more samples
    % near the center.
    plot[id=asymptotic-example] function{(t*t*t)*sin(1/(t*t*t)),(t*t*t)*cos(1/(t*t*t))}
    node[right] {$\bigl(x(t),y(t)\bigr) = (t\sin \frac{1}{t}, t\cos \frac{1}{t})$};

  \fill[red] (0.63662,0) circle [radius=2pt]
    node [below right,fill=white,yshift=-4pt] {$(\frac{2}{\pi},0)$};
\end{tikzpicture}
```

99 Design Principles

This section describes the basic layer of PGF. This layer is built on top of the system layer. Whereas the system layer just provides the absolute minimum for drawing graphics, the basic layer provides numerous commands that make it possible to create sophisticated graphics easily and also quickly.

The basic layer does not provide a convenient syntax for describing graphics, which is left to frontends like Ti*k*Z. For this reason, the basic layer is typically used only by "other programs". For example, the BEAMER package uses the basic layer extensively, but does not need a convenient input syntax. Rather, speed and flexibility are needed when BEAMER creates graphics.

The following basic design principles underlie the basic layer:

1. Structuring into a core and modules.
2. Consistently named TEX macros for all graphics commands.
3. Path-centered description of graphics.
4. Coordinate transformation system.

99.1 Core and Modules

The basic layer consists of a *core package*, called `pgfcore`, which provides the most basic commands, and several *modules* like commands for plotting (in the `plot` module). Modules are loaded using the `\usepgfmodule` command.

If you say `\usepackage{pgf}` or `\input{pgf.tex}` or `\usemodule[pgf]`, the `plot` and `shapes` modules are preloaded (as well as the core and the system layer).

99.2 Communicating with the Basic Layer via Macros

In order to "communicate" with the basic layer you use long sequences of commands that start with `\pgf`. You are only allowed to give these commands inside a `{pgfpicture}` environment. (Note that `{tikzpicture}` opens a `{pgfpicture}` internally, so you can freely mix PGF commands and Ti*k*Z commands inside a `{tikzpicture}`.) It is possible to "do other things" between the commands. For example, you might use one command to move to a certain point, then have a complicated computation of the next point, and then move there.



```
\newdimen\myypos
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{\myypos}}
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \advance \myypos by 1cm
  \pgfpathlineto{\pgfpoint{1cm}{\myypos}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following naming conventions are used in the basic layer:

1. All commands and environments start with `pgf`.
2. All commands that specify a point (a coordinate) start with `\pgfpoint`.
3. All commands that extend the current path start with `\pgfpath`.
4. All commands that set/change a graphics parameter start with `\pgfset`.
5. All commands that use a previously declared object (like a path, image or shading) start with `\pgfuse`.
6. All commands having to do with coordinate transformations start with `\pgftransform`.
7. All commands having to do with arrow tips start with `\pgfarrows`.
8. All commands for "quickly" extending or drawing a path start with `\pgfpathq` or `\pgfusepathq`.
9. All commands having to do with matrices start with `\pgfmatrix`.

99.3 Path-Centered Approach

In PGF the most important entity is the *path*. All graphics are composed of numerous paths that can be stroked, filled, shaded, or clipped against. Paths can be closed or open, they can self-intersect and consist of unconnected parts.

Paths are first *constructed* and then *used*. In order to construct a path, you can use commands starting with `\pgfpath`. Each time such a command is called, the current path is extended in some way.

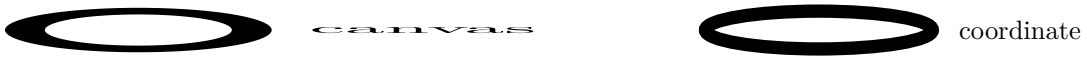
Once a path has been completely constructed, you can use it using the command `\pgfusepath`. Depending on the parameters given to this command, the path will be stroked (drawn) or filled or subsequent drawings will be clipped against this path.

99.4 Coordinate Versus Canvas Transformations

PGF provides two transformation systems: PGF's own *coordinate* transformation matrix and PDF's or PostScript's *canvas* transformation matrix. These two systems are quite different. Whereas a scaling by a factor of, say, 2 of the canvas causes *everything* to be scaled by this factor (including the thickness of lines and text), a scaling of two in the coordinate system causes only the *coordinates* to be scaled, but not the line width nor text.

By default, all transformations only apply to the coordinate transformation system. However, using the command `\pgflowlevel` it is possible to apply a transformation to the canvas.

Coordinate transformations are often preferable over canvas transformations. Text and lines that are transformed using canvas transformations suffer from differing sizes and lines whose thickness differs depending on whether the line is horizontal or vertical. To appreciate the difference, consider the following two "circles" both of which have been scaled in the x-direction by a factor of 3 and by a factor of 0.5 in the y-direction. The left circle uses a canvas transformation, the right uses PGF's coordinate transformation (some viewers will render the left graphic incorrectly since they do no apply the low-level transformation the way they should):



100 Hierarchical Structures: Package, Environments, Scopes, and Text

100.1 Overview

PGF uses two kinds of hierarchical structuring: First, the package itself is structured hierarchically, consisting of different packages that are built on top of each other. Second, PGF allows you to structure your graphics hierarchically using environments and scopes.

100.1.1 The Hierarchical Structure of the Package

The PGF system consists of several layers:

System layer. The lowest layer is called the *system layer*, though it might also be called "driver layer" or perhaps "backend layer". Its job is to provide an abstraction of the details of which driver is used to transform the `.dvi` file. The system layer is implemented by the package `pgfsys`, which will load appropriate driver files as needed.

The system layer is documented in Part X.

Basic layer. The basic layer is loaded by the package `pgfcore` and subsequent use of the command `\usepgfmodule` to load additional modules of the basic layer.

The basic layer is documented in the present part.

Frontend layer. The frontend layer is not loaded by a single package. Rather, different packages, like Ti*k*Z or PGFPICT2E, are different frontends to the basic layer.

The Ti*k*Z frontend is documented in Part III.

Each layer will automatically load the necessary files of the layers below it.

In addition to the packages of these layers, there are also some library packages. These packages provide additional definitions of things like new arrow tips or new plot handlers.

The library packages are documented in Part V.

100.1.2 The Hierarchical Structure of Graphics

Graphics in PGF are typically structured hierarchically. Hierarchical structuring can be used to identify groups of graphical elements that are to be treated "in the same way". For example, you might group together a number of paths, all of which are to be drawn in red. Then, when you decide later on that you like them to be drawn in, say, blue, all you have to do is to change the color once.

The general mechanism underlying hierarchical structuring is known as *scoping* in computer science. The idea is that all changes to the general "state" of the graphic that are done inside a scope are local to that scope. So, if you change the color inside a scope, this does not affect the color used outside the scope. Likewise, when you change the line width in a scope, the line width outside is not changed, and so on.

There are different ways of starting and ending scopes of graphic parameters. Unfortunately, these scopes are sometimes "in conflict" with each other and it is sometimes not immediately clear which scopes apply. In essence, the following scoping mechanisms are available:

1. The "outermost" scope supported by PGF is the `{pgfpicture}` environment. All changes to the graphic state done inside a `{pgfpicture}` are local to that picture.

   In general, it is *not* possible to set graphic parameters globally outside any `{pgfpicture}` environments. Thus, you can *not* say `\pgfsetlinewidth{1pt}` at the beginning of your document to have a default line width of one point. Rather, you have to (re)set all graphic parameters inside each `{pgfpicture}`. (If this is too bothersome, try defining some macro that does the job for you.)

2. Inside a `{pgfpicture}` you can use a `{pgfscope}` environment to keep changes of the graphic state local to that environment.

   The effect of commands that change the graphic state are local to the current `{pgfscope}`, but not always to the current TEX group. Thus, if you open a TEX group (some text in curly braces) inside a `{pgfscope}`, and if you change, for example, the dash pattern, the effect of this changed dash pattern will persist till the end of the `{pgfscope}`.

Unfortunately, this is not always the case. *Some* graphic parameters only persist till the end of the current TEX group. For example, when you use `\pgfsetarrows` to set the arrow tip inside a TEX group, the effect lasts only till the end of the current TEX group.

3. Some graphic parameters are not scoped by `{pgfscope}` but "already" by TEX groups. For example, the effect of coordinate transformation commands is always local to the current TEX group.

   Since every `{pgfscope}` automatically creates a TEX group, all graphic parameters that are local to the current TEX group are also local to the current `{pgfscope}`.

4. Some graphic parameters can only be scoped using TEX groups, since in some situations it is not possible to introduce a `{pgfscope}`. For example, a path always has to be completely constructed and used in the same `{pgfscope}`. However, we might wish to have different coordinate transformations apply to different points on the path. In this case, we can use TEX groups to keep the effect local, but we could not use `{pgfscope}`.

5. The `\pgftext` command can be used to create a scope in which TEX "escapes back" to normal TEX mode. The text passed to the `\pgftext` is "heavily guarded" against having any effect on the scope in which it is used. For example, it is possible to use another `{pgfpicture}` environment inside the argument of `\pgftext`.

Most of the complications can be avoided if you stick to the following rules:

- Give graphic commands only inside `{pgfpicture}` environments.
- Use `{pgfscope}` to structure graphics.
- Do not use TEX groups inside graphics, *except* for keeping the effect of coordinate transformations local.

100.2 The Hierarchical Structure of the Package

Before we come to the structuring commands provided by PGF to structure your graphics, let us first have a look at the structure of the package itself.

100.2.1 The Core Package

To use PGF, include the following package:

```
\usepackage{pgfcore} % LaTeX
\input{pgfcore.tex} % plain TeX
\usemodule[pgfcore]  % ConTeXt
```

This package loads the complete core of the "basic layer" of PGF, but not any modules. That is, it will load all of the commands described in the current part of this manual, but it will not load frontends like Ti*k*Z. It will also load the system layer. To load additional modules, use the `\usepgfmodule` command explained below.

The following package is just a convenience.

```
\usepackage{pgf} % LaTeX
\input{pgf.tex} % plain TeX
\usemodule[pgf]  % ConTeXt
```

This package loads the `pgfcore` and the two modules `shapes` and `plot`.

In LaTeX, the package takes two options:

`\usepackage[draft]{pgf}`

When this option is set, all images will be replaced by empty rectangles. This can speedup compilation.

`\usepackage[version=`⟨*version*⟩`]{pgf}`

Indicates that the commands of version ⟨*version*⟩ need to be defined. If you set ⟨*version*⟩ to `0.65`, then a large bunch of "compatibility commands" are loaded. If you set ⟨*version*⟩ to `0.96`, then these compatibility commands will not be loaded.

If this option is not given at all, then the commands of all versions are defined.

100.2.2 The Modules

\usepgfmodule{⟨*module names*⟩}

Once the core has been loaded, you can use this command to load further modules. The modules in the ⟨*module names*⟩ list should be separated by commas. Instead of curly braces, you can also use square brackets, which is something ConTEXt users will like. If you try to load a module a second time, nothing will happen.

Example: `\usepgfmodule{matrix,shapes}`

What this command does is to load the file `pgfmodule`⟨*module*⟩`.code.tex` for each ⟨*module*⟩ in the list of ⟨*module names*⟩. Thus, to write your own module, all you need to do is to place a file of the appropriate name somewhere TEX can find it. LATEX, plain TEX, and ConTEXt users can then use your library.

The following modules are available for use with `pgfcore`:

- The `plot` module provides commands for plotting functions. The commands are explained in Section 112.
- The `shapes` module provides commands for drawing shapes and nodes. These commands are explained in Section 106.
- The `decorations` module provides commands for adding decorations to paths. These commands are explained in Section 103.
- The `matrix` module provides the `\pgfmatrix` command. The commands are documented in Section 107.

100.2.3 The Library Packages

There is a special command for loading library packages. The difference between a library and module is the following: A library just defines additional objects using the basic layer, whereas a module adds completely new functionality. For instance, a `decorations` library defines additional decorations, while a decoration module defines the whole code for handling decorations.

\usepgflibrary{⟨*list of libraries*⟩}

Use this command to load further libraries. The list of libraries should contain the names of libraries separated by commas. Instead of curly braces, you can also use square brackets. If you try to load a library a second time, nothing will happen.

Example: `\usepgflibrary{arrows}`

This command causes the file `pgflibrary`⟨*library*⟩`.code.tex` to be loaded for each ⟨*library*⟩ in the ⟨*list of libraries*⟩. This means that in order to write your own library file, place a file of the appropriate name somewhere where TEX can find it. LATEX, plain TEX, and ConTEXt users can then use your library.

You should also consider adding a Ti*k*Z library that simply includes your PGF library.

100.3 The Hierarchical Structure of the Graphics

100.3.1 The Main Environment

Most, but not all, commands of the PGF package must be given within a `{pgfpicture}` environment. The only commands that (must) be given outside are commands having to do with including images (like `\pgfuseimage`) and with inserting complete shadings (like `\pgfuseshading`). However, just to keep life entertaining, the `\pgfshadepath` command must be given *inside* a `{pgfpicture}` environment.

\begin{pgfpicture}
⟨*environment contents*⟩
\end{pgfpicture}

This environment will insert a TEX box containing the graphic drawn by the ⟨*environment contents*⟩ at the current position.

The size of the bounding box. The size of the box is determined in the following manner: While PGF parses the ⟨*environment contents*⟩, it keeps track of a bounding box for the graphic. Essentially, this bounding box is the smallest box that contains all coordinates mentioned in the graphics. Some coordinates may be "mentioned" by PGF itself; for example, when you add circle to the current path, the support points of the curve making up the circle are also "mentioned" despite the fact that you will not "see" them in your code.

Once the ⟨*environment contents*⟩ have been parsed completely, a TEX box is created whose size is the size of the computed bounding box and this box is inserted at the current position.

Hello ▭ World!

```
Hello \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} World!
```

Sometimes, you may need more fine-grained control over the size of the bounding box. For example, the computed bounding box may be too large or you intentionally wish the box to be "too small". In these cases, you can use the command `\pgfusepath{use as bounding box}`, as described in Section 104.6.

The baseline of the bounding box. When the box containing the graphic is inserted into the normal text, the baseline of the graphic is normally at the bottom of the graphic. For this reason, the following two sets of code lines have the same effect, despite the fact that the second graphic uses "higher" coordinates than the first:

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
\end{pgfpicture}.
```

You can change the baseline using the `\pgfsetbaseline` command, see below.

Rectangles ▭ and ▭.

```
Rectangles \begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture} and \begin{pgfpicture}
  \pgfpathrectangle{\pgfpoint{0ex}{1ex}}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{stroke}
  \pgfsetbaseline{0pt}
\end{pgfpicture}.
```

Including text and images in a picture. You cannot directly include text and images in a picture. Thus, you should *not* simply write some text in a `{pgfpicture}` or use a command like `\includegraphics` or even `\pgfimage`. In all these cases, you need to place the text inside a `\pgftext` command. This will "escape back" to normal TEX mode, see Section 100.3.3 for details.

Remembering a picture position for later reference. After a picture has been typeset, its position on the page is normally forgotten by PGF and also by TEX. This means that is not possible to reference a node in this picture later on. In particular, it is normally impossible to draw lines between nodes in different pictures automatically.

In order to make PGF "remember" a picture, the TEX-if `\ifpgfrememberpicturepositiononpage` should be set to `true`. It is only important that this TEX-if is `true` at the end of the `{pgfpicture}`-environment, so you can switch it on inside the environment. However, you can also just switch it on globally, then the positions of all pictures are remembered.

There are several reasons why the remembering is not switched on by default. First, it does not work for all backend drivers (currently, it works only for pdfTEX). Second, it requires two passes of TEX over the file; on the first pass all positions will be wrong. Third, for every remembered picture a line is added to the `.aux`-file, which may result in a large number of extra lines.

Despite all these "problems", for documents that are processed with pdfTEX and in which there is only a small number of pictures (less than a hundred or so), you can switch on this option globally, it will not cause any significant slowing of TEX.

\pgfpicture
  ⟨*environment contents*⟩
\endpgfpicture

The plain TEX version of the environment. Note that in this version, also, a TEX group is created around the environment.

\startpgfpicture
  ⟨*environment contents*⟩
\stoppgfpicture

The ConTEXt version of the environment.

\ifpgfrememberpicturepositiononpage

Determines whether the position of pictures on the page should be recorded. The value of this TEX-if at the end of a `{pgfpicture}` environment is important, not the value at the beginning.

If this option is set to true of a picture, PGF will attempt to record the position of the picture on the page. (This attempt will fail with most drivers and when it works, it typically requires two runs of TEX.) The position is not directly accessible. Rather, the nodes mechanism will use this position if you access a node from another picture. See Sections 106.3.2 and 17.13 for more details.

\pgfsetbaseline{⟨*dimension*⟩}

This command specifies a y-coordinate of the picture that should be used as the baseline of the whole picture. When a PGF picture has been typeset completely, PGF must decide at which height the baseline of the picture should lie. Normally, the baseline is set to the y-coordinate of the bottom of the picture, but it is often desirable to use a different height.

Text◯,◯,◯,◯.

```
Text
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}
\end{pgfpicture},
\begin{pgfpicture}
  \pgfsetbaseline{0pt}
  \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}
\end{pgfpicture},
\begin{pgfpicture}
  \pgfsetbaseline{.5ex}
  \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}
\end{pgfpicture},
\begin{pgfpicture}
  \pgfsetbaseline{-1ex}
  \pgfpathcircle{\pgfpointorigin}{1ex}\pgfusepath{stroke}
\end{pgfpicture}.
```

\pgfsetbaselinepointnow{⟨*point*⟩}

This command specifies the baseline indirectly, namely as the y-coordinate that the given ⟨*point*⟩ has when the command is called.

\pgfsetbaselinepointlater{⟨*point*⟩}

This command also specifies the baseline indirectly, but the y-coordinate of the given ⟨*point*⟩ is only computed at the end of the picture.

Hello ~~world.~~

```
\usetikzlibrary {shapes.misc}
Hello
\begin{pgfpicture}
  \pgfsetbaselinepointlater{\pgfpointanchor{X}{base}}
  % Note: no shape X, yet
  \pgfnode{cross out}{center}{world.}{X}{\pgfusepath{stroke}}
\end{pgfpicture}
```

100.3.2 Graphic Scope Environments

Inside a `{pgfpicture}` environment you can substructure your picture using the following environment:

```
\begin{pgfscope}
  ⟨environment contents⟩
\end{pgfscope}
```

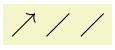
All changes to the graphic state done inside this environment are local to the environment. The graphic state includes the following:

- The line width.
- The stroke and fill colors.
- The dash pattern.
- The line join and cap.
- The miter limit.
- The canvas transformation matrix.
- The clipping path.

Other parameters may also influence how graphics are rendered, but they are *not* part of the graphic state. For example, the arrow tip kind is not part of the graphic state and the effect of commands setting the arrow tip kind are local to the current TEX group, not to the current `{pgfscope}`. However, since `{pgfscope}` starts and ends a TEX group automatically, a `{pgfscope}` can be used to limit the effect of, say, commands that set the arrow tip kind.



```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetlinewidth{2pt}
      \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathrectangle{\pgfpoint{3ex}{0ex}}{\pgfpoint{2ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathrectangle{\pgfpoint{6ex}{0ex}}{\pgfpoint{2ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \begin{pgfscope}
    {
      \pgfsetarrows{->}
      \pgfpathmoveto{\pgfpointorigin}\pgfpathlineto{\pgfpoint{2ex}{2ex}}
      \pgfusepath{stroke}
    }
    \pgfpathmoveto{\pgfpoint{3ex}{0ex}}\pgfpathlineto{\pgfpoint{5ex}{2ex}}
    \pgfusepath{stroke}
  \end{pgfscope}
  \pgfpathmoveto{\pgfpoint{6ex}{0ex}}\pgfpathlineto{\pgfpoint{8ex}{2ex}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

At the start of the scope, the current path must be empty, that is, you cannot open a scope while constructing a path.

It is usually a good idea *not* to introduce TEX groups inside a `{pgfscope}` environment.

```
\pgfscope
  ⟨environment contents⟩
\endpgfscope
```

The plain TEX version of the environment.

```
\startpgfscope
  ⟨environment contents⟩
\stoppgfscope
```

The ConTEXt version of the environment.

The following scopes also encapsulate certain properties of the graphic state. However, they are typically not used directly by the user.

```
\begin{pgfinterruptpath}
  ⟨environment contents⟩
\end{pgfinterruptpath}
```

This environment can be used to temporarily interrupt the construction of the current path. The effect will be that the path currently under construction will be "stored away" and restored at the end of the environment. Inside the environment you can construct a new path and do something with it.

An example application of this environment is the arrow tip caching. Suppose you ask PGF to use a specific arrow tip kind. When the arrow tip needs to be rendered for the first time, PGF will "cache" the path that makes up the arrow tip. To do so, it interrupts the current path construction and then protocols the path of the arrow tip. The `{pgfinterruptpath}` environment is used to ensure that this does not interfere with the path to which the arrow tips should be attached.

This command does *not* install a `{pgfscope}`. In particular, it does not call any `\pgfsys@` commands at all, which would, indeed, be dangerous in the middle of a path construction.

```
\pgfinterruptpath
  ⟨environment contents⟩
\endpgfinterruptpath
```

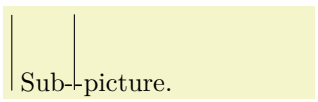
The plain TEX version of the environment.

```
\startpgfinterruptpath
  ⟨environment contents⟩
\stoppgfinterruptpath
```

The ConTEXt version of the environment.

```
\begin{pgfinterruptpicture}
  ⟨environment contents⟩
\end{pgfinterruptpicture}
```

This environment can be used to temporarily interrupt a `{pgfpicture}`. However, the environment is intended only to be used at the beginning and end of a box that is (later) inserted into a `{pgfpicture}` using `\pgfqbox`. You cannot use this environment directly inside a `{pgfpicture}`.




```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}} % In the middle of path, now
  \newbox\mybox
  \setbox\mybox=\hbox{
    \begin{pgfinterruptpicture}
      Sub-\begin{pgfpicture} % a subpicture
        \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
        \pgfpathlineto{\pgfpoint{1cm}{1cm}}
        \pgfusepath{stroke}
      \end{pgfpicture}-picture.
    \end{pgfinterruptpicture}
  }
  \pgfqbox{\mybox}%
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}\hskip3.9cm
```

```
\pgfinterruptpicture
  ⟨environment contents⟩
\endpgfinterruptpicture
```

The plain TEX version of the environment.

```
\startpgfinterruptpicture
  ⟨environment contents⟩
\stoppgfinterruptpicture
```

The ConTEXt version of the environment.

```
\begin{pgfinterruptboundingbox}
  ⟨environment contents⟩
\end{pgfinterruptboundingbox}
```

This environment temporarily interrupts the computation of the bounding box and sets up a new bounding box. At the beginning of the environment the old bounding box is saved and an empty bounding box is installed. After the environment the original bounding box is reinstalled as if nothing has happened.

```
\pgfinterruptboundingbox
  ⟨environment contents⟩
\endpgfinterruptboundingbox
```

The plain TEX version of the environment.

```
\startpgfinterruptboundingbox
  ⟨environment contents⟩
\stoppgfinterruptboundingbox
```

The ConTEXt version of the environment.

100.3.3 Inserting Text and Images

Often, you may wish to add normal TEX text at a certain point inside a `{pgfpicture}`. You cannot do so "directly", that is, you cannot simply write this text inside the `{pgfpicture}` environment. Rather, you must pass the text as an argument to the `\pgftext` command.

You must *also* use the `\pgftext` command to insert an image or a shading into a `{pgfpicture}`.

`\pgftext[⟨options⟩]{⟨text⟩}`

This command will typeset ⟨*text*⟩ in normal TEX mode and insert the resulting box into the `{pgfpicture}`. The bounding box of the graphic will be updated so that all of the text box is inside. By default, the text box is centered at the origin, but this can be changed either by giving appropriate ⟨*options*⟩ or by applying an appropriate coordinate transformation beforehand.

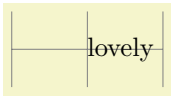
The ⟨*text*⟩ may contain verbatim text. (In other words, the ⟨*text*⟩ "argument" is not a normal argument, but is put in a box and some `\aftergroup` hackery is used to find the end of the box.)

PGF's current (high-level) coordinate transformation is synchronized with the canvas transformation matrix temporarily when the text box is inserted. The effect is that if there is currently a high-level rotation of, say, 30 degrees, the ⟨*text*⟩ will also be rotated by thirty degrees. If you do not want this effect, you have to (possibly temporarily) reset the high-level transformation matrix.

The ⟨*options*⟩ keys are used with the path `/pgf/text/`. The following keys are defined for this path:

`/pgf/text/left` (no value)

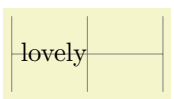
The key causes the text box to be placed such that its left border is on the origin.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[left] {lovely}}
```

`/pgf/text/right` (no value)

The key causes the text box to be placed such that its right border is on the origin.



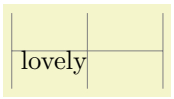

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
     \pgftext[right] {lovely}}
```

`/pgf/text/top` (no value)

This key causes the text box to be placed such that its top is on the origin. This option can be used together with the `left` or `right` option.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[top] {lovely}}
```



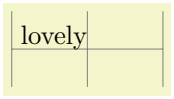

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[top,right] {lovely}}
```

/pgf/text/bottom (no value)

This key causes the text box to be placed such that its bottom is on the origin.



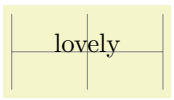

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[bottom] {lovely}}
```



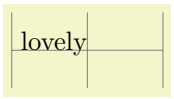

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[bottom,right] {lovely}}
```

/pgf/text/base (no value)

This key causes the text box to be placed such that its baseline is on the origin.



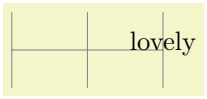

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base] {lovely}}
```




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,right] {lovely}}
```

/pgf/text/at=⟨*point*⟩ (no default)

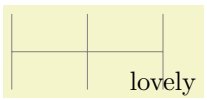
Translates the origin (that is, the point where the text is shown) to ⟨*point*⟩.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,at={\pgfpoint{1cm}{0cm}}] {lovely}}
```

/pgf/text/x=⟨*dimension*⟩ (no default)

Translates the origin by ⟨*dimension*⟩ along the x-axis.



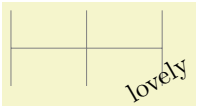

```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,x=1cm,y=-0.5cm] {lovely}}
```

/pgf/text/y=⟨*dimension*⟩ (no default)

This key works like the `x` option.

/pgf/text/rotate=⟨*degree*⟩ (no default)

Rotates the coordinate system by ⟨*degree*⟩. This will also rotate the text box.




```
\tikz{\draw[help lines] (-1,-.5) grid (1,.5);
    \pgftext[base,x=1cm,y=-0.5cm,rotate=30] {lovely}}
```

100.4 Object Identifiers

Graphical objects can have an *identifier,* which allows you to reference the object later on. For instance, you could reference the object as the target of a hyperlink (although this capability is not necessarily implemented by drivers) or as the target of an animation; indeed, animations always need an object identifier to identify the to-be-animated object.

Attaching an identifier to an object is a two-step process:

1. You call `\pgfuseid{`⟨*id*⟩`}` to choose an id, which is a normal string.
2. Next, you call one of several commands like `\pgfidscope` or `\pgftext`, which create an object. This object will have then have the id.

100.4.1 Commands for Creating Graphic Objects

The following system level commands create an object with an id:

1. `\pgfsys@begin@idscope`, which creates a graphic scope.
2. `\pgfsys@viewboxmeet` or `\pgfsys@viewboxslice`, which create view boxes,
3. `\pgfsys@fill`, `\pgfsys@stroke`, and all other path usage command,
4. `\pgfsys@hbox` or `\pgfsys@hboxsynced`, which create text boxes, and
5. `\pgfsys@animate...`, which create animations.

These system layer commands are, in turn, called by the following basic layer commands (and, also, by the commands that call them, in turn):

- `\pgfidscope`, which creates an id scope (see below).
- `\pgfviewboxscope`, which creates a view box.
- `\pgfusepath`, which creates a path.
- `\pgftext` and `\pgfnode` and `\pgfmultipartnode`, which create text boxes and nodes, and
- `\pgfanimateattribute`, which creates an animation.

`\begin{pgfidscope}`
⟨*environment contents*⟩
`\end{pgfidscope}`

Creates a graphic scope that will have the id last used with `\pgfuseid` attached to, provided such an id was set and was not already used with another object. In the latter cases, no graphic scope is created. Thus, if you wish to ensure that a graphic scope is created, you must (additionally) call `\pgfscope` inside or outside the id scope.

`\pgfidscope`
⟨*environment contents*⟩
`\endpgfidscope`

The plain TEX version of the environment.

`\startpgfidscope`
⟨*environment contents*⟩
`\stoppgfidscope`

The ConTEXt version of the environment.

100.4.2 Settings and Querying Identifiers

In order to attach an identifier to an object, you first use the following command:

\pgfuseid{⟨*name*⟩}

The ⟨*name*⟩ is a string by which the object will be referenced (see `\pgfidrefnextuse`). The next time a graphic object is created in the current TEX scope, the name will be attached to it (actually, it will get a system layer identifier attached to it that is automatically created using `\pgfsys@new@id`, the ⟨*name*⟩ is bound to that identifier and it can be retrieved using `\pgfidrefnextuse`). This holds true only for the next object: If a second object is created, it will not get the name attached to it. This does not mean, however, that you cannot attach the same name to different objects; you just need to call `\pgfuseid` again before each object.

Besides the ⟨*name*⟩ (or, more precisely, besides the system layer identifier it refers to), the current *identifier type* is also important: Actually, a graphic object is not referenced by a system layer identifier, but by the combination of the identifier and a type. You can use the following commands for modifying the type used for the creation of objects:

\pgfusetype{⟨*type*⟩}

Sets the type used for the referencing of graphic objects for the current scope to ⟨*type*⟩ or, if ⟨*type*⟩ starts with a dot, appends ⟨*type*⟩ to the current type.

You use this command with compound graphic objects: Before each part of a graphic object, set the type to an appropriate value. Now, if the object is named using `\pgfuseid`, you can later on access all parts of the compound object using the combination of the ⟨*name*⟩ used with `\pgfuseid` and the type of the part.

As an example, this system is used to give you access to the different parts of a node: When you say `\pgfuseid{mynode}` and then create a node, you can use `mynode` with the empty type to reference the graphics scope that encompasses the whole node, but also `mynode` together with the type `background` to access the background path of the node.

In detail, PGF uses this command to set the following types:

- Inside the command `\pgfviewboxscope`, the type `.view` is used for the view object.
- Inside the command `\pgfmultipartnode`, the type `.behind background` is used for the scope of drawings behind the background. Similarly, `.before background` and `.behind foreground` and finally `.before foreground` are used with the respective parts of a node.
- Also inside a node, `.background` and `.foreground` are used as types of the background and foreground paths, respectively.
- Finally, inside a node, for each text part, the text part's name is used as a type (so `.text` is used for the main part).

In addition, Ti*k*Z uses this command in the following situations:

- The type `.path` is used with a named path (named using the `name` key). This is the graphic object you need to reference when you wish to morph a path.
- The type `.path picture` is used with the scope of the optional path picture.
- The type `.path fill` is used with the path used for filling. This is not the same as the normal path in case the path is filled and patterned, for instance.
- The type `.path shade` is used with the path used for shading a path.

\pgfpushtype

Pushes the current type on an internal global stack. The idea is to allow you to temporarily change the current type without having to open a TEX scope.

\pgfpoptype

Restores the most recent type from the internal global stack of types.

\pgfclearid

Clears the current id (and type) for the local scope.

\pgfidrefnextuse{⟨*macro*⟩}{⟨*name*⟩}

This command assigns a system layer identifier (the identifier returned by \pgfsys@new@id) to the ⟨*macro*⟩, namely the one that will be used the *next* time \pgfuseid is used. You use this command for "forward referencing".

A typical use case is the following: A key like whom for animations uses this command to get the system identifier that will be used for a future object. Then, this identifier can be passed to system layer commands like \pgfsys@animation@whom.

Note that the "next" use need not be on the same page (or there may not even be any use at all), in which case the reference will not refer to any object.

\pgfidrefprevuse{⟨*macro*⟩}{⟨*name*⟩}

Works like \pgfidrefnextuse, only it references the most recent *previous* use of the ⟨*name*⟩. As for \pgfidrefnextuse, the most recent use need not be on the same page.

\pgfaliasid{⟨*alias*⟩}{⟨*name*⟩}

Creates an alias of a name inside the current TEX scope. After calling this command, you can use ⟨*alias*⟩ anywhere where you would normally use ⟨*name*⟩. Note that the binding between ⟨*alias*⟩ and ⟨*name*⟩ is not kept when \pgfuseid is used on the ⟨*name*⟩ (or the ⟨*alias*⟩).

\pgfgaliasid{⟨*1*⟩}{⟨*2*⟩}

Like \pgfaliasid, only the alias is set globally.

\pgfifidreferenced{⟨*name*⟩}{⟨*then code*⟩}{⟨*else code*⟩}

If ⟨*name*⟩ has been referenced, ⟨*then code*⟩ is executed, otherwise ⟨*else code*⟩.

100.5 Resource Description Framework Annotations (RDFa)

With certain output formats (in particular, with SVG) you can insert annotations into the output file following the standard set by the *resource description framework* (known as "RDF", please consult the literature on RDF and RDFa for an introduction to resource descriptions and ontologies and their purpose in general). To do so, you call one (or several) of the following commands before you call \pgfidscope. The attributes and values you specify using the commands will then be added to the resulting scope (if the driver supports this, which is only the case for SVG at the moment). As an example, when you write

```
\pgfrdfresource{/fruits/apple}
\pgfidscope
...
\pgfendidscope
```

in the resulting SVG file you get

```
<g resource="/fruits/apple">
  ...
</g>
```

Most of the following commands just set a single attribute for the next id scope. In some cases, however, repeated calling of these commands makes sense and causes the passed values to accumulate as in the following example:

```
\pgfrdfresource{/fruits/apple}
\pgfrdfproperty{http://foo.com/props/juicy}
\pgfrdfproperty{http://foo.com/props/green}
\pgfidscope
...
\pgfendidscope
```

Now you get:

```
<g resource="/fruits/apple"
   property="http://foo.com/props/juicy http://foo.com/props/green">
  ...
</g>
```

The following commands "accumulate": \pgfrdfproperty, \pgfrdfrel, \pgfrdfrev and also the command \pgfrdftypeof.

\pgfrdfabout{⟨*text*⟩}

Adds the RDF attribute `about="`⟨*text*⟩`"` to the next id scope (please see the RDFa specification for details on the semantics of `about` in the context of the resource description framework).

The following commands work the same way.

\pgfrdfcontent{⟨*text*⟩}

\pgfrdfdatatype{⟨*text*⟩}

\pgfrdfhref{⟨*text*⟩}

\pgfrdfinlist

\pgfrdfprefix{⟨*text*⟩}

\pgfrdfproperty{⟨*text*⟩}

\pgfrdfrel{⟨*text*⟩}

\pgfrdfresource{⟨*text*⟩}

\pgfrdfrev{⟨*text*⟩}

\pgfrdfsrc{⟨*text*⟩}

\pgfrdftypeof{⟨*text*⟩}

\pgfrdfvocab{⟨*text*⟩}

100.6 Error Messages and Warnings

Sometimes, a command inside PGF may fail. In this case, two commands are useful to communicate with the author:

\pgferror{⟨*message*⟩}

Stops the processing of the current document and prints out the ⟨*message*⟩. In LaTeX, this will be done using `\PackageError`, otherwise `\errmessage` is used directly.

\pgfwarning{⟨*message*⟩}

Prints the ⟨*message*⟩ on the output, but does not interrupt the processing. In LaTeX, this will be done using `\PackageWarning`, otherwise a write to stream 17 is used.

101 Specifying Coordinates

101.1 Overview

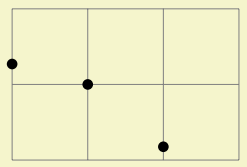
Most PGF commands expect you to provide the coordinates of a *point* (also called *coordinate*) inside your picture. Points are always "local" to your picture, that is, they never refer to an absolute position on the page, but to a position inside the current `{pgfpicture}` environment. To specify a coordinate you can use commands that start with `\pgfpoint`.

101.2 Basic Coordinate Commands

The following commands are the most basic for specifying a coordinate.

`\pgfpoint{`⟨*x coordinate*⟩`}{`⟨*y coordinate*⟩`}`

Yields a point location. The coordinates are given as TEX dimensions.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{1cm}{1cm}} {2pt}
  \pgfpathcircle{\pgfpoint{2cm}{5pt}} {2pt}
  \pgfpathcircle{\pgfpoint{0pt}{.5in}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

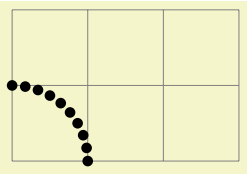
`\pgfpointorigin`

Yields the origin. Same as `\pgfpoint{0pt}{0pt}`.

`\pgfpointpolar{`⟨*degree*⟩`}{`⟨*radius*⟩`/`⟨*y-radius*⟩`}`

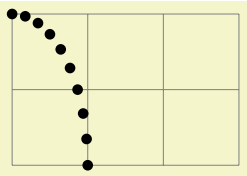
Yields a point location given in polar coordinates. You can specify the angle only in degrees, radians are not supported, currently.

If the optional ⟨*y-radius*⟩ is given, the polar coordinate is actually a coordinate on an ellipse whose x-radius is given by ⟨*radius*⟩ and whose y-radius is given by ⟨*y-radius*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

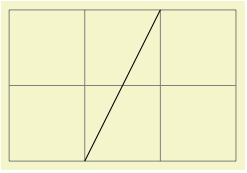
  \foreach \angle in {0,10,...,90}
    {\pgfpathcircle{\pgfpointpolar{\angle}{1cm and 2cm}}{2pt}}
  \pgfusepath{fill}
\end{tikzpicture}
```

101.3 Coordinates in the XY-Coordinate System

Coordinates can also be specified as multiples of an x-vector and a y-vector. Normally, the x-vector points one centimeter in the x-direction and the y-vector points one centimeter in the y-direction, but using the commands `\pgfsetxvec` and `\pgfsetyvec` they can be changed. Note that the x- and y-vector do not necessarily point "horizontally" and "vertically".

`\pgfpointxy{`⟨s_x⟩`}{`⟨s_y⟩`}`

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector.

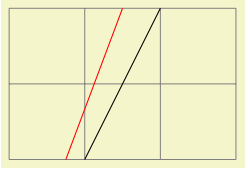


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetxvec{⟨*point*⟩}

Sets that current x-vector for usage in the xyz-coordinate system.

Example:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}

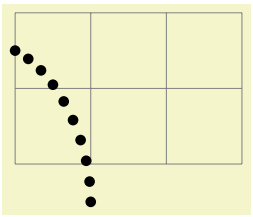
  \color{red}
  \pgfsetxvec{\pgfpoint{0.75cm}{0cm}}
  \pgfpathmoveto{\pgfpointxy{1}{0}}
  \pgfpathlineto{\pgfpointxy{2}{2}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfsetyvec{⟨*point*⟩}

Works like `\pgfsetxvec`.

\pgfpointpolarxy{⟨*degree*⟩}{⟨*radius*⟩/⟨*y-radius*⟩}

This command is similar to the `\pgfpointpolar` command, but the ⟨*radius*⟩ is now a factor to be interpreted in the xy-coordinate system. This means that a degree of `0` is the same as the x-vector of the xy-coordinate system times ⟨*radius*⟩ and a degree of `90` is the y-vector times ⟨*radius*⟩. As for `\pgfpointpolar`, a ⟨*radius*⟩ can also be a pair separated by a slash. In this case, the x- and y-vectors are multiplied by different factors.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

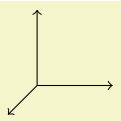
  \begin{scope}[x={(1cm,-5mm)},y=1.5cm]
    \foreach \angle in {0,10,...,90}
      {\pgfpathcircle{\pgfpointpolarxy{\angle}{1}}{2pt}}
    \pgfusepath{fill}
  \end{scope}
\end{tikzpicture}
```

101.4 Three Dimensional Coordinates

It is also possible to specify a point as a multiple of three vectors, the x-, y-, and z-vector. This is useful for creating simple three dimensional graphics.

\pgfpointxyz{⟨s_x⟩}{⟨s_y⟩}{⟨s_z⟩}

Yields a point that is situated at s_x times the x-vector plus s_y times the y-vector plus s_z times the z-vector.



```
\begin{pgfpicture}
  \pgfsetarrowsend{to}

  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{0}{1}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{0}{1}{0}}
  \pgfusepath{stroke}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpointxyz{1}{0}{0}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetzvec{⟨*point*⟩}

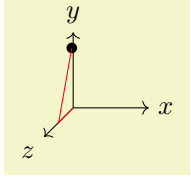
Works like \pgfsetxvec.

Inside the xyz-coordinate system, you can also specify points using spherical and cylindrical coordinates.

\pgfpointcylindrical{⟨*degree*⟩}{⟨*radius*⟩}{⟨*height*⟩}

This command yields the same as

```
\pgfpointadd{\pgfpointpolarxy{degree}{radius}}{\pgfpointxyz{0}{0}{height}}
```



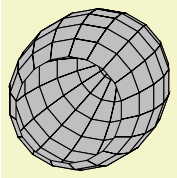

```
\begin{tikzpicture}
  \draw [->] (0,0) -- (1,0,0) node [right] {$x$};
  \draw [->] (0,0) -- (0,1,0) node [above] {$y$};
  \draw [->] (0,0) -- (0,0,1) node [below left] {$z$};

  \pgfpathcircle{\pgfpointcylindrical{80}{1}{.5}}{2pt}
  \pgfusepath{fill}

  \draw[red] (0,0) -- (0,0,.5) -- +(80:1);
\end{tikzpicture}
```

\pgfpointspherical{⟨*longitude*⟩}{⟨*latitude*⟩}{⟨*radius*⟩}

This command yields a point "on the surface of the earth" specified by the ⟨*longitude*⟩ and the ⟨*latitude*⟩. The radius of the earth is given by ⟨*radius*⟩. The equator lies in the xy-plane.



```
\begin{tikzpicture}
  \pgfsetfillcolor{lightgray}

  \foreach \latitude in {-90,-75,...,30}
  {
    \foreach \longitude in {0,20,...,360}
    {
      \pgfpathmoveto{\pgfpointspherical{\longitude}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude+20}{\latitude+15}{1}}
      \pgfpathlineto{\pgfpointspherical{\longitude}{\latitude+15}{1}}
      \pgfpathclose
    }
    \pgfusepath{fill,stroke}
  }
\end{tikzpicture}
```

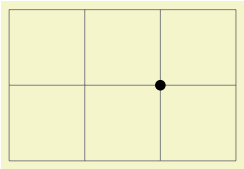
101.5 Building Coordinates From Other Coordinates

Many commands allow you to construct a coordinate in terms of other coordinates.

101.5.1 Basic Manipulations of Coordinates

\pgfpointadd{⟨v_1⟩}{⟨v_2⟩}

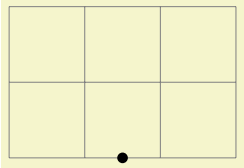
Returns the sum vector $\langle v_1 \rangle + \langle v_2 \rangle$.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointadd{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointscale{⟨*factor*⟩}{⟨*coordinate*⟩}

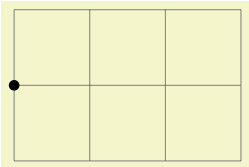
Returns the vector ⟨*factor*⟩⟨*coordinate*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointscale{1.5}{\pgfpoint{1cm}{0cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointdiff{⟨*start*⟩}{⟨*end*⟩}

Returns the difference vector ⟨*end*⟩ − ⟨*start*⟩.

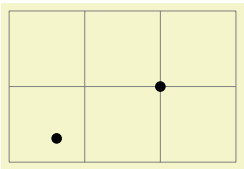


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpointdiff{\pgfpoint{1cm}{0cm}}{\pgfpoint{1cm}{1cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointnormalised{⟨*point*⟩}

This command returns a normalised version of ⟨*point*⟩, that is, a vector of length 1pt pointing in the direction of ⟨*point*⟩. If ⟨*point*⟩ is the 0-vector or extremely short, a vector of length 1pt pointing upwards is returned.

This command is *not* implemented by calculating the length of the vector, but rather by calculating the angle of the vector and then using (something equivalent to) the `\pgfpointpolar` command. This ensures that the point will really have length 1pt, but it is not guaranteed that the vector will *precisely* point in the direction of ⟨*point*⟩ due to the fact that the polar tables are accurate only up to one degree. Normally, this is not a problem.



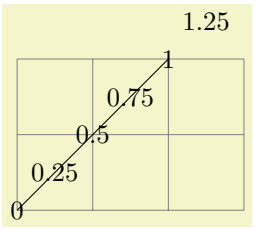
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointscale{20}
    {\pgfpointnormalised{\pgfpoint{2cm}{1cm}}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

101.5.2 Points Traveling along Lines and Curves

The commands in this section allow you to specify points on a line or a curve. Imagine a point "traveling" along a curve from some point p to another point q. At time $t = 0$ the point is at p and at time $t = 1$ it is at q and at time, say, $t = 1/2$ it is "somewhere in the middle". The exact location at time $t = 1/2$ will not necessarily be the "halfway point", that is, the point whose distance on the curve from p and q is equal. Rather, the exact location will depend on the "speed" at which the point is traveling, which in turn depends on the lengths of the support vectors in a complicated manner. If you are interested in the details, please see a good book on Bézier curves.

\pgfpointlineattime{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point q*⟩}

Yields a point that is the tth fraction between p and q, that is, $p + t(q - p)$. For $t = 1/2$ this is the middle of p and q.

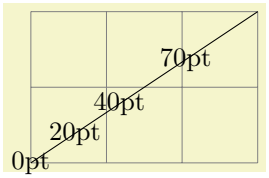



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,...,1.25}
    {\pgftext[at=
      \pgfpointlineattime{\t}{\pgfpointorigin}{\pgfpoint{2cm}{2cm}}]{\t}}
\end{tikzpicture}
```

\pgfpointlineatdistance{⟨*distance*⟩}{⟨*start point*⟩}{⟨*end point*⟩}

Yields a point that is located ⟨*distance*⟩ many units away from the start point in the direction of the end point. In other words, this is the point that results if we travel ⟨*distance*⟩ steps from ⟨*start point*⟩ towards ⟨*end point*⟩.

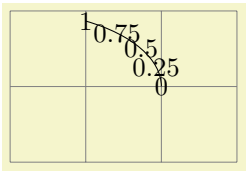
Example:




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \d in {0pt,20pt,40pt,70pt}
    {\pgftext[at=
      \pgfpointlineatdistance{\d}{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}]{\d}}
\end{tikzpicture}
```

\pgfpointarcaxesattime{⟨*time t*⟩}{⟨*center*⟩}{⟨*0-degree axis*⟩}{⟨*90-degree axis*⟩}{⟨*start angle*⟩}{⟨*end angle*⟩}

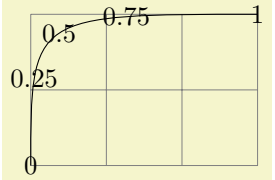
Yields a point on the arc between ⟨*start angle*⟩ and ⟨*end angle*⟩ on an ellipse whose center is at ⟨*center*⟩ and whose two principal axes are ⟨*0-degree axis*⟩ and ⟨*90-degree axis*⟩. For $t = 0$ the point at the ⟨*start angle*⟩ is returned and for $t = 1$ the point at the ⟨*end angle*⟩.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}}
  \pgfpatharcaxes{0}{60}{\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=\pgfpointarcaxesattime{\t}{\pgfpoint{0cm}{1cm}}
       {\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}{0}{60}]{\t}}
\end{tikzpicture}
```

\pgfpointcurveattime{⟨*time t*⟩}{⟨*point p*⟩}{⟨*point s_1*⟩}{⟨*point s_2*⟩}{⟨*point q*⟩}

Yields a point that is on the Bézier curve from p to q with the support points s_1 and s_2. The time t is used to determine the location, where $t = 0$ yields p and $t = 1$ yields q.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
  \foreach \t in {0,0.25,0.5,0.75,1}
    {\pgftext[at=\pgfpointcurveattime{\t}{\pgfpointorigin}
                                      {\pgfpoint{0cm}{2cm}}
                                      {\pgfpoint{0cm}{2cm}}
                                      {\pgfpoint{3cm}{2cm}}]{\t}}
\end{tikzpicture}
```

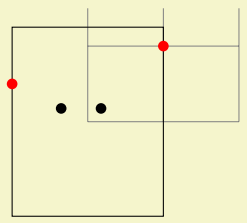
101.5.3 Points on Borders of Objects

The following commands are useful for specifying a point that lies on the border of special shapes. They are used, for example, by the shape mechanism to determine border points of shapes.

\pgfpointborderrectangle{⟨*direction point*⟩}{⟨*corner*⟩}

This command returns a point that lies on the intersection of a line starting at the origin and going towards the point ⟨*direction point*⟩ and a rectangle whose center is in the origin and whose upper right corner is at ⟨*corner*⟩.

The ⟨*direction point*⟩ should have length "about 1pt", but it will be normalized automatically. Nevertheless, the "nearer" the length is to 1pt, the less rounding errors.

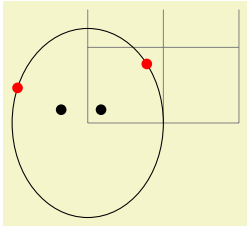


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathrectanglecorners{\pgfpoint{-1cm}{-1.25cm}}{\pgfpoint{1cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderrectangle
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

\pgfpointborderellipse{⟨*direction point*⟩}{⟨*corner*⟩}

This command works like the corresponding command for rectangles, only this time the ⟨*corner*⟩ is the corner of the bounding rectangle of an ellipse.



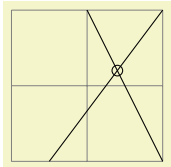
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,1.5);
  \pgfpathellipse{\pgfpointorigin}{\pgfpoint{1cm}{0cm}}{\pgfpoint{0cm}{1.25cm}}
  \pgfusepath{stroke}

  \pgfpathcircle{\pgfpoint{5pt}{5pt}}{2pt}
  \pgfpathcircle{\pgfpoint{-10pt}{5pt}}{2pt}
  \pgfusepath{fill}
  \color{red}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{5pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfpathcircle{\pgfpointborderellipse
    {\pgfpoint{-10pt}{5pt}}{\pgfpoint{1cm}{1.25cm}}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

101.5.4 Points on the Intersection of Lines

\pgfpointintersectionoflines{⟨*p*⟩}{⟨*q*⟩}{⟨*s*⟩}{⟨*t*⟩}

This command returns the intersection of a line going through p and q and a line going through s and t. If the lines do not intersection, an arithmetic overflow will occur.

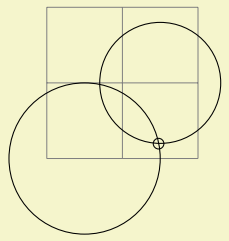


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \draw (.5,0) -- (2,2);
  \draw (1,2) -- (2,0);
  \pgfpathcircle{%
    \pgfpointintersectionoflines
      {\pgfpointxy{.5}{0}}{\pgfpointxy{2}{2}}
      {\pgfpointxy{1}{2}}{\pgfpointxy{2}{0}}}
    {2pt}
  \pgfusepath{stroke}
\end{tikzpicture}
```

101.5.5 Points on the Intersection of Two Circles

\pgfpointintersectionofcircles{⟨p_1⟩}{⟨p_2⟩}{⟨r_1⟩}{⟨r_2⟩}{⟨*solution*⟩}

This command returns the intersection of the two circles centered at p_1 and p_2 with radii r_1 and r_2. If ⟨*solution*⟩ is `1`, the first intersection is returned, otherwise the second one is returned.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (2,2);
  \draw (0.5,0) circle [radius=1];
  \draw (1.5,1) circle [radius=.8];
  \pgfpathcircle{%
    \pgfpointintersectionofcircles
      {\pgfpointxy{.5}{0}}{\pgfpointxy{1.5}{1}}
      {1cm}{0.8cm}{1}}
    {2pt}
  \pgfusepath{stroke}
\end{tikzpicture}
```

101.5.6 Points on the Intersection of Two Paths

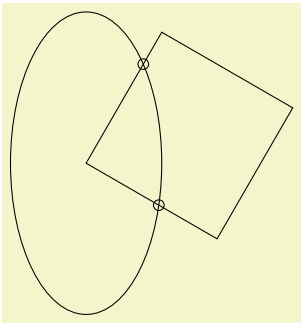
Ti*k*Z Library `intersections`

```
\usepgflibrary{intersections}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[intersections]  % ConTeXt when using pure PGF
\usetikzlibrary{intersections} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[intersections] % ConTeXt when using TikZ
```

This library defines the below command and allows you to calculate the intersections of two arbitrary paths. However, due to the low accuracy of TEX, the paths should not be "too complicated". In particular, you should not try to intersect paths consisting of lots of very small segments such as plots or decorated paths.

\pgfintersectionofpaths{⟨*path 1*⟩}{⟨*path 2*⟩}

This command finds the intersection points on the paths ⟨*path 1*⟩ and ⟨*path 2*⟩. The number of intersection points ("solutions") that are found will be stored, and each point can be accessed afterward. The code for ⟨*path 1*⟩ and ⟨*path 2*⟩ is executed within a TEX group and so can contain transformations (which will be in addition to any existing transformations). The code should not use the path in any way, unless the path is saved first and restored afterward. PGF will regard solutions as "a bit special", in that the points returned will be "absolute" and unaffected by any further transformations.



```
\usetikzlibrary {intersections}
\begin{pgfpicture}
\pgfintersectionofpaths
{
  \pgfpathellipse{\pgfpointxy{0}{0}}{\pgfpointxy{1}{0}}{\pgfpointxy{0}{2}}
  \pgfgetpath\temppath
  \pgfusepath{stroke}
  \pgfsetpath\temppath
}
{
  \pgftransformrotate{-30}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpointxy{2}{2}}
  \pgfgetpath\temppath
  \pgfusepath{stroke}
  \pgfsetpath\temppath
}
\foreach \s in {1,...,\pgfintersectionsolutions}
  {\pgfpathcircle{\pgfpointintersectionsolution{\s}}{2pt}}
\pgfusepath{stroke}
\end{pgfpicture}
```

\pgfintersectionsolutions

After using the `\pgfintersectionofpaths` command, this TEX-macro will indicate the number of solutions found.

\pgfpointintersectionsolution{⟨*number*⟩}

After using the \pgfintersectionofpaths command, this command will return the point for solution ⟨*number*⟩ or the origin if this solution was not found. By default, the intersections are simply returned in the order that the intersection algorithm finds them. Unfortunately, this is not necessarily a "helpful" ordering. However the following two commands can be used to order the solutions more helpfully.

\pgfintersectionsortbyfirstpath

Using this command will mean the solutions will be sorted along ⟨*path 1*⟩.

\pgfintersectionsortbysecondpath

Using this command will mean the solutions will be sorted along ⟨*path 2*⟩.

101.6 Extracting Coordinates

There are two commands that can be used to "extract" the x- or y-coordinate of a coordinate.

\pgfextractx{⟨*dimension*⟩}{⟨*point*⟩}

Sets the TEX-⟨*dimension*⟩ to the x-coordinate of the point.

```
\newdimen\mydim
\pgfextractx{\mydim}{\pgfpoint{2cm}{4pt}}
%% \mydim is now 2cm
```

\pgfextracty{⟨*dimension*⟩}{⟨*point*⟩}

Like \pgfextractx, except for the y-coordinate.

\pgfgetlastxy{⟨*macro for x*⟩}{⟨*macro for y*⟩}

Stores the most recently used (x, y) coordinates into two macros.

Macro x is '56.9055pt' and macro y is '113.81102pt'.

```
\pgfpoint{2cm}{4cm}
\pgfgetlastxy{\macrox}{\macroy}
Macro $x$ is `\macrox' and macro $y$ is `\macroy'.
```

Since (x, y) coordinates are usually assigned globally, it is safe to use this command after path operations.

101.7 Internals of How Point Commands Work

As a normal user of PGF you do not need to read this section. It is relevant only if you need to understand how the point commands work internally.

When a command like \pgfpoint{1cm}{2pt} is called, all that happens is that the two TEX-dimension variables \pgf@x and \pgf@y are set to 1cm and 2pt, respectively. These variables belong to the set of internal PGF registers, see section 117 for details. A command like \pgfpathmoveto that takes a coordinate as parameter will just execute this parameter and then use the values of \pgf@x and \pgf@y as the coordinates to which it will move the pen on the current path.

Since commands like \pgfpointnormalised modify other variables besides \pgf@x and \pgf@y during the computation of the final values of \pgf@x and \pgf@y, it is a good idea to enclose a call of a command like \pgfpoint in a TEX-scope and then make the changes of \pgf@x and \pgf@y global as in the following example:

```
...
{ % open scope
  \pgfpointnormalised{\pgfpoint{1cm}{1cm}}
  \global\pgf@x=\pgf@x % make the change of \pgf@x persist past the scope
  \global\pgf@y=\pgf@y % make the change of \pgf@y persist past the scope
}
% \pgf@x and \pgf@y are now set correctly, all other variables are
% unchanged
```

Since this situation arises very often, the macro \pgf@process can be used to perform the above code:

\pgf@process{⟨*code*⟩}

Executes the ⟨*code*⟩ in a scope and then makes \pgf@x and \pgf@y global.

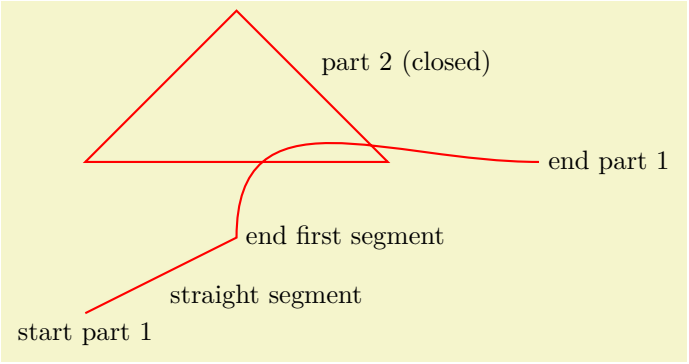
Note that this macro is used often internally. For this reason, it is not a good idea to keep anything important in the variables \pgf@x and \pgf@y since they will be overwritten and changed frequently. Instead, intermediate values can be stored in the TEX-dimensions \pgf@xa, \pgf@xb, \pgf@xc and their y-counterparts \pgf@ya, \pgf@yb, \pgf@yc. For example, here is the code of the command \pgfpointadd:

```
\def\pgfpointadd#1#2{%
  \pgf@process{#1}%
  \pgf@xa=\pgf@x%
  \pgf@ya=\pgf@y%
  \pgf@process{#2}%
  \advance\pgf@x by\pgf@xa%
  \advance\pgf@y by\pgf@ya}
```

102 Constructing Paths

102.1 Overview

The "basic entity of drawing" in PGF is the *path*. A path consists of several parts, each of which is either a closed or open curve. An open curve has a starting point and an end point and, in between, consists of several *segments*, each of which is either a straight line or a Bézier curve. Here is an example of a path (in red) consisting of two parts, one open, one closed:




```
\begin{tikzpicture}[scale=2]
  \draw[thick,red]
       (0,0) coordinate (a)
    -- coordinate (ab) (1,.5) coordinate (b)
    .. coordinate (bc) controls +(up:1cm) and +(left:1cm) .. (3,1)  coordinate (c)
       (0,1) -- (2,1) -- coordinate (x) (1,2) -- cycle;

  \draw (a)  node[below] {start part 1}
        (ab) node[below right] {straight segment}
        (b)  node[right] {end first segment}
        (c)  node[right] {end part 1}
        (x)  node[above right]  {part 2 (closed)};
\end{tikzpicture}
```

A path, by itself, has no "effect", that is, it does not leave any marks on the page. It is just a set of points on the plane. However, you can *use* a path in different ways. The most natural actions are *stroking* (also known as *drawing*) and *filling*. Stroking can be imagined as picking up a pen of a certain diameter and "moving it along the path". Filling means that everything "inside" the path is filled with a uniform color. Naturally, the open parts of a path must first be closed before a path can be filled.

In PGF, there are numerous commands for constructing paths, all of which start with `\pgfpath`. There are also commands for *using* paths, though most operations can be performed by calling `\pgfusepath` with an appropriate parameter.

As a side-effect, the path construction commands keep track of two bounding boxes. One is the bounding box for the current path, the other is a bounding box for all paths in the current picture. See Section 102.13 for more details.

Each path construction command extends the current path in some way. The "current path" is a global entity that persists across TEX groups. Thus, between calls to the path construction commands you can perform arbitrary computations and even open and close TEX groups. The current path only gets "flushed" when the `\pgfusepath` command is called (or when the soft-path subsystem is used directly, see Section 121).

102.2 The Move-To Path Operation

The most basic operation is the move-to operation. It must be given at the beginning of paths, though some path construction command (like `\pgfpathrectangle`) generate move-tos implicitly. A move-to operation can also be used to start a new part of a path.

`\pgfpathmoveto{`⟨*coordinate*⟩`}`

This command expects a PGF-coordinate like `\pgfpointorigin` as its parameter. When the current path is empty, this operation will start the path at the given ⟨*coordinate*⟩. If a path has already been partly constructed, this command will end the current part of the path and start a new one.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}} % New part
  \pgfpathlineto{\pgfpoint{3cm}{0.5cm}}
  \pgfpathlineto{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

It will update the bounding box of the current path and picture, if necessary.

102.3 The Line-To Path Operation

\pgfpathlineto{⟨*coordinate*⟩}

This command extends the current path in a straight line to the given ⟨*coordinate*⟩. If this command is given at the beginning of path without any other path construction command given before (in particular without a move-to operation), the TEX file may compile without an error message, but a viewer application may display an error message when trying to render the picture.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{2cm}{1cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

It will update the bounding box of the current path and picture, if necessary.

102.4 The Curve-To Path Operations

\pgfpathcurveto{⟨*support 1*⟩}{⟨*support 2*⟩}{⟨*coordinate*⟩}

This command extends the current path with a Bézier curve from the last point of the path to ⟨*coordinate*⟩. The ⟨*support 1*⟩ and ⟨*support 2*⟩ are the first and second support point of the Bézier curve. For more information on Bézier curves, please consult a standard textbook on computer graphics.

Like the line-to command, this command may not be the first path construction command in a path.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathcurveto
    {\pgfpoint{1cm}{1cm}}{\pgfpoint{2cm}{1cm}}{\pgfpoint{3cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply the current coordinate transformation matrix to ⟨*coordinate*⟩ before using it.

It will update the bounding box of the current path and picture, if necessary. However, the bounding box is simply made large enough such that it encompasses all of the support points and the ⟨*coordinate*⟩. This will guarantee that the curve is completely inside the bounding box, but the bounding box will

typically be quite a bit too large. It is not clear (to me) how this can be avoided without resorting to "some serious math" in order to calculate a precise bounding box.

\pgfpathquadraticcurveto{⟨*support*⟩}{⟨*coordinate*⟩}

This command works like \pgfpathcurveto, only it uses a quadratic Bézier curve rather than a cubic one. This means that only one support point is needed.



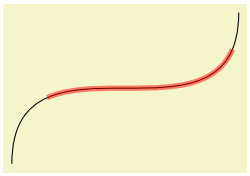
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathquadraticcurveto
    {\pgfpoint{1cm}{1cm}}{\pgfpoint{2cm}{0cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

Internally, the quadratic curve is converted into a cubic curve. The only noticeable effect of this is that the points used for computing the bounding box are the control points of the converted curve rather than ⟨*support*⟩. The main effect of this is that the bounding box will be a bit tighter than might be expected. In particular, ⟨*support*⟩ will not always be part of the bounding box.

There exist two commands to draw only part of a cubic Bézier curve:

\pgfpathcurvebetweentime{⟨*time* t_1⟩}{⟨*time* t_2⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

This command draws the part of the curve described by p, s_1, s_2 and q between the times t_1 and t_2. A time value of 0 indicates the point p and a time value of 1 indicates point q. This command includes a moveto operation to the first point.



```
\begin{tikzpicture}
  \draw [thin] (0,0) .. controls (0,2) and (3,0) .. (3,2);
  \pgfpathcurvebetweentime{0.25}{0.9}{\pgfpointxy{0}{0}}{\pgfpointxy{0}{2}}
    {\pgfpointxy{3}{0}}{\pgfpointxy{3}{2}}
  \pgfsetstrokecolor{red}
  \pgfsetstrokeopacity{0.5}
  \pgfsetlinewidth{2pt}
  \pgfusepath{stroke}
\end{tikzpicture}
```

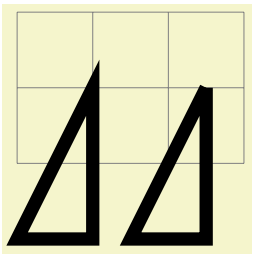
\pgfpathcurvebetweentimecontinue{⟨*time* t_1⟩}{⟨*time* t_2⟩}{⟨*point* p⟩}{⟨*point* s_1⟩}{⟨*point* s_2⟩}{⟨*point* q⟩}

This command works like \pgfpathcurvebetweentime, except that a moveto operation is *not* made to the first point.

102.5 The Close Path Operation

\pgfpathclose

This command closes the current part of the path by appending a straight line to the start point of the current part. Note that there *is* a difference between closing a path and using the line-to operation to add a straight line to the start of the current path. The difference is demonstrated by the upper corners of the triangles in the following example:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{5pt}
  \pgfpathmoveto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{0cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{-1cm}}
  \pgfpathclose
  \pgfpathmoveto{\pgfpoint{2.5cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{-1cm}}
  \pgfpathlineto{\pgfpoint{2.5cm}{1cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

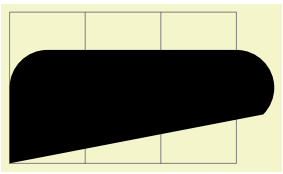
102.6 Arc, Ellipse and Circle Path Operations

The path construction commands that we have discussed up to now are sufficient to create all paths that can be created "at all". However, it is useful to have special commands to create certain shapes, like circles, that arise often in practice.

In the following, the commands for adding (parts of) (transformed) circles to a path are described.

`\pgfpatharc{`⟨*start angle*⟩`}{`⟨*end angle*⟩`}{`⟨*radius*⟩ `and` ⟨*y-radius*⟩`}`

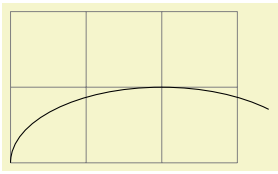
This command appends a part of a circle (or an ellipse) to the current path. Imagine the curve between ⟨*start angle*⟩ and ⟨*end angle*⟩ on a circle of radius ⟨*radius*⟩ (if ⟨*start angle*⟩ < ⟨*end angle*⟩, the curve goes around the circle counterclockwise, otherwise clockwise). This curve is now moved such that the point where the curve starts is the previous last point of the path. Note that this command will *not* start a new part of the path, which is important for example for filling purposes.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{1cm}}
  \pgfpatharc{180}{90}{.5cm}
  \pgfpathlineto{\pgfpoint{3cm}{1.5cm}}
  \pgfpatharc{90}{-45}{.5cm}
  \pgfusepath{fill}
\end{tikzpicture}
```

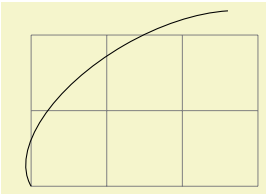
Saying `\pgfpatharc{0}{360}{1cm}` "nearly" gives you a full circle. The "nearly" refers to the fact that the circle will not be closed. You can close it using `\pgfpathclose`.

If the optional ⟨*y-radius*⟩ is given, the ⟨*radius*⟩ is the x-radius and the ⟨*y-radius*⟩ the y-radius of the ellipse from which the curve is taken:



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm and 1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The axes of the circle or ellipse from which the arc is "taken" always point up and right. However, the current coordinate transformation matrix will have an effect on the arc. This can be used to, say, rotate an arc:



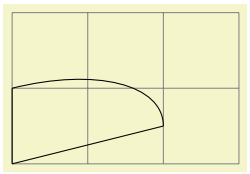
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharc{180}{45}{2cm and 1cm}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will update the bounding box of the current path and picture, if necessary. Unless rotation or shearing transformations are applied, the bounding box will be tight.

`\pgfpatharcaxes{`⟨*start angle*⟩`}{`⟨*end angle*⟩`}{`⟨*first axis*⟩`}{`⟨*second axis*⟩`}`

This command is similar to `\pgfpatharc`. The main difference is how the ellipse or circle is specified from which the arc is taken. The two parameters ⟨*first axis*⟩ and ⟨*second axis*⟩ are the 0°-axis and the 90°-axis of the ellipse from which the path is taken. Thus, `\pgfpatharc{0}{90}{1cm and 2cm}` has the same effect as

```
\pgfpatharcaxes{0}{90}{\pgfpoint{1cm}{0cm}}{\pgfpoint{0cm}{2cm}}
```

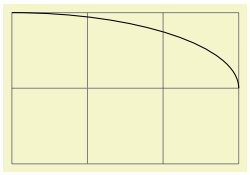


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw (0,0) -- (2cm,5mm) (0,0) -- (0cm,1cm);

  \pgfpathmoveto{\pgfpoint{2cm}{5mm}}
  \pgfpatharcaxes{0}{90}{\pgfpoint{2cm}{5mm}}{\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

\pgfpatharcto{⟨*x-radius*⟩}{⟨*y-radius*⟩}{⟨*rotation*⟩} {⟨*large arc flag*⟩}{⟨*counterclockwise flag*⟩} {⟨*target point*⟩}

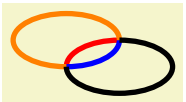
This command (which directly corresponds to the arc-path command of SVG) is used to add an arc to the path that starts at the current point and ends at ⟨*target point*⟩. This arc is part of an ellipse that is determined in the following way: Imagine an ellipse with radii ⟨*x-radius*⟩ and ⟨*y-radius*⟩ that is rotated around its center by ⟨*rotation*⟩ degrees. When you move this ellipse around in the plane, there will be exactly two positions such that the two current point and the target point lie on the border of the ellipse (excluding pathological cases). The flags ⟨*large arc flag*⟩ and ⟨*clockwise flag*⟩ are then used to decide which of these ellipses should be picked and which arc on the picked ellipsis should be used.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfpathmoveto{\pgfpoint{0mm}{20mm}}
  \pgfpatharcto{3cm}{1cm}{0}{0}{0}{\pgfpoint{3cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

Both flags are considered to be false exactly if they evaluate to 0, otherwise they are true. If the ⟨*large arc flag*⟩ is true, then the angle spanned by the arc will be greater than 180°, otherwise it will be less than 180°. The ⟨*clockwise flag*⟩ is used to determine which of the two ellipses should be used: if the flag is true, then the arc goes from the current point to the target point in a counterclockwise direction, otherwise in a clockwise fashion.



```
\begin{tikzpicture}
  \pgfsetlinewidth{2pt}
  % Flags 0 0: red
  \pgfsetstrokecolor{red}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 0 1: blue
  \pgfsetstrokecolor{blue}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{0}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 1 0: orange
  \pgfsetstrokecolor{orange}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{0}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
  % Flags 1 1: black
  \pgfsetstrokecolor{black}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpatharcto{20pt}{10pt}{0}{1}{1}{\pgfpoint{20pt}{10pt}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

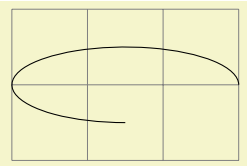
Warning: The internal computations necessary for this command are numerically very unstable. In particular, the arc will not always really end at the ⟨*target coordinate*⟩, but may be off by up to several points. A more precise positioning is currently infeasible due to TEX's numerical weaknesses. The only case it works quite nicely is when the resulting angle is a multiple of 90°.

\pgfpatharctoprecomputed{⟨*center point*⟩}{⟨*start angle*⟩}{⟨*end angle*⟩}{⟨*end point*⟩} {⟨*x-radius*⟩}{⟨*y-radius*⟩}{⟨*ratio x-radius/y-radius*⟩}{⟨*ratio y-radius/x-radius*⟩}

A specialized arc operation which is fast and numerically stable, provided a lot of information is given in advance.

In contrast to \pgfpatharc, it explicitly interpolates start and end points.

In contrast to \pgfpatharcto, this routine is numerically stable and quite fast since it relies on a lot of available information.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \def\cx{1.5cm}% center x
  \def\cy{1cm}% center y
  \def\startangle{0}%
  \def\endangle{270}%
  \def\a{1.5cm}% xradius
  \def\b{0.5cm}% yradius
  \pgfmathparse{\a/\b}\let\abratio=\pgfmathresult
  \pgfmathparse{\b/\a}\let\baratio=\pgfmathresult
  %
  % start point:
  \pgfpathmoveto{\pgfpoint{\cx+\a*cos(\startangle)}{\cy+\b*sin(\startangle)}}%
  \pgfpatharctoprecomputed
    {\pgfpoint{\cx}{\cy}}
    {\startangle}
    {\endangle}
    {\pgfpoint{\cx+\a*cos(\endangle)}{\cy+\b*sin(\endangle)}}% end point
    {\a}
    {\b}
    {\abratio}
    {\baratio}
  \pgfusepath{draw}
\end{tikzpicture}
```

\pgfpatharctomaxstepsize

The quality of arc approximation taken by \pgfpatharctoprecomputed by means of Bézier splines is controlled by a mesh width, which is initially

\def\pgfpatharctoprecomputed{45}.

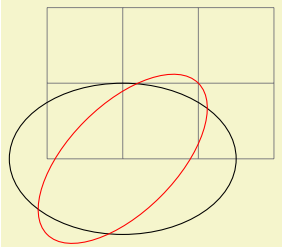
The mesh width is provided in (full!) degrees. The smaller the mesh width, the more precise the arc approximation.

Use an empty value to disable spline approximation (uses a single cubic polynomial for the complete arc).

The value must be an integer!

\pgfpathellipse{⟨*center*⟩}{⟨*first axis*⟩}{⟨*second axis*⟩}

The effect of this command is to append an ellipse to the current path (if the path is not empty, a new part is started). The ellipse's center will be ⟨*center*⟩ and ⟨*first axis*⟩ and ⟨*second axis*⟩ are the axis *vectors*. The same effect as this command can also be achieved using an appropriate sequence of move-to, arc, and close operations, but this command is easier and faster.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1.5cm}{0cm}}
                 {\pgfpoint{0cm}{1cm}}
  \pgfusepath{draw}
  \color{red}
  \pgfpathellipse{\pgfpoint{1cm}{0cm}}
                 {\pgfpoint{1cm}{1cm}}
                 {\pgfpoint{-0.5cm}{0.5cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations to all coordinates of the ellipse. However, the coordinate transformations are applied only after the ellipse is "finished conceptually". Thus, a transformation of 1cm to the right will simply shift the ellipse one centimeter to the right; it will not add 1cm to the x-coordinates of the two axis vectors.

The command will update the bounding box of the current path and picture, if necessary.

\pgfpathcircle{⟨*center*⟩}{⟨*radius*⟩}

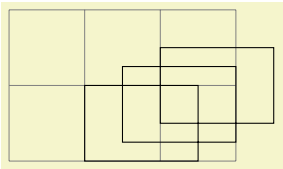
A shorthand for \pgfpathellipse applied to ⟨*center*⟩ and the two axis vectors (⟨*radius*⟩, 0) and (0, ⟨*radius*⟩).

102.7 Rectangle Path Operations

Another shape that arises frequently is the rectangle. Two commands can be used to add a rectangle to the current path. Both commands will start a new part of the path.

\pgfpathrectangle{⟨*corner*⟩}{⟨*diagonal vector*⟩}

Adds a rectangle to the path whose one corner is ⟨*corner*⟩ and whose opposite corner is given by ⟨*corner*⟩ + ⟨*diagonal vector*⟩.

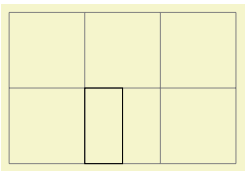


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectangle{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1.5cm}{0.25cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{2cm}{0.5cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

\pgfpathrectanglecorners{⟨*corner*⟩}{⟨*opposite corner*⟩}

Adds a rectangle to the path whose two opposing corners are ⟨*corner*⟩ and ⟨*opposite corner*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathrectanglecorners{\pgfpoint{1cm}{0cm}}{\pgfpoint{1.5cm}{1cm}}
  \pgfusepath{draw}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes tightly.

102.8 The Grid Path Operation

\pgfpathgrid[⟨*options*⟩]{⟨*first corner*⟩}{⟨*second corner*⟩}

Appends a grid to the current path. That is, a (possibly large) number of parts are added to the path, each part consisting of a single horizontal or vertical straight line segment.

Conceptually, the origin is part of the grid and the grid is clipped to the rectangle specified by the ⟨*first corner*⟩ and the ⟨*second corner*⟩. However, no clipping occurs (this command just adds parts to the current path) and the points where the lines enter and leave the "clipping area" are computed and used to add simple lines to the current path.

The following keys influence the grid:

/pgf/stepx=⟨*dimension*⟩ (no default, initially 1cm)

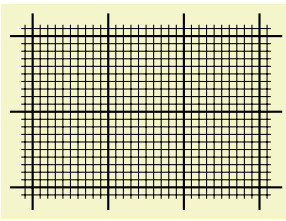
The horizontal stepping.

/pgf/stepy=⟨*dimension*⟩ (no default, initially 1cm)

The vertical stepping.

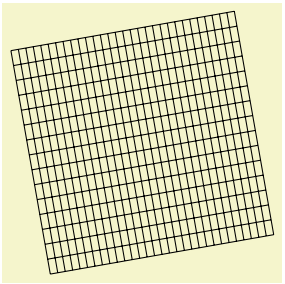
/pgf/step=⟨*vector*⟩ (no default)

Sets the horizontal stepping to the x-coordinate of ⟨*vector*⟩ and the vertical stepping to its y-coordinate.



```
\begin{pgfpicture}
  \pgfsetlinewidth{0.8pt}
  \pgfpathgrid[step={\pgfpoint{1cm}{1cm}}]
    {\pgfpoint{-3mm}{-3mm}}{\pgfpoint{33mm}{23mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{0.4pt}
  \pgfpathgrid[stepx=1mm,stepy=1mm]
    {\pgfpoint{-1.5mm}{-1.5mm}}{\pgfpoint{31.5mm}{21.5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes. As for ellipses, the transformations are applied to the "conceptually finished" grid.



```
\begin{pgfpicture}
  \pgftransformrotate{10}
  \pgfpathgrid[stepx=1mm,stepy=2mm]{\pgfpoint{0mm}{0mm}}{\pgfpoint{30mm}{30mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

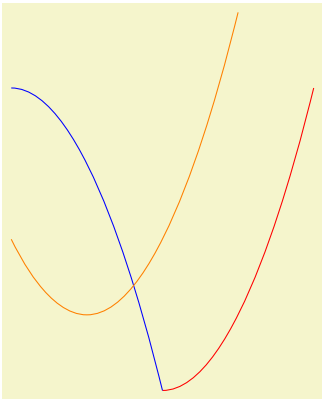
102.9 The Parabola Path Operation

\pgfpathparabola{⟨*bend vector*⟩}{⟨*end vector*⟩}

This command appends two half-parabolas to the current path. The first starts at the current point and ends at the current point plus ⟨*bend vector*⟩. At this point, it has its bend. The second half parabola starts at that bend point and ends at point that is given by the bend plus ⟨*end vector*⟩.

If you set ⟨*end vector*⟩ to the null vector, you append only a half parabola that goes from the current point to the bend; by setting ⟨*bend vector*⟩ to the null vector, you append only a half parabola that goes through the current point and ⟨*end vector*⟩ and has its bend at the current point.

It is not possible to use this command to draw a part of a parabola that does not contain the bend.



```
\begin{pgfpicture}
  % Half-parabola going ``up and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpointorigin}{\pgfpoint{2cm}{4cm}}
  \color{red}
  \pgfusepath{stroke}

  % Half-parabola going ``down and right''
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathparabola{\pgfpoint{-2cm}{4cm}}{\pgfpointorigin}
  \color{blue}
  \pgfusepath{stroke}

  % Full parabola
  \pgfpathmoveto{\pgfpoint{-2cm}{2cm}}
  \pgfpathparabola{\pgfpoint{1cm}{-1cm}}{\pgfpoint{2cm}{4cm}}
  \color{orange}
  \pgfusepath{stroke}
\end{pgfpicture}
```

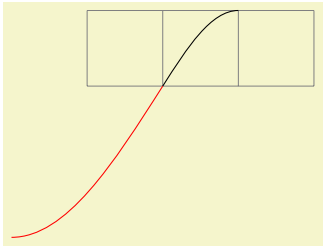
The command will apply coordinate transformations and update the bounding boxes.

102.10 Sine and Cosine Path Operations

Sine and cosine curves often need to be drawn and the following commands may help with this. However, they only allow you to append sine and cosine curves in intervals that are multiples of $\pi/2$.

\pgfpathsine{⟨*vector*⟩}

This command appends a sine curve in the interval $[0, \pi/2]$ to the current path. The sine curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩.



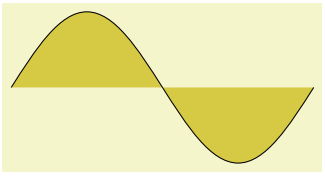
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,1);
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}

  \color{red}
  \pgfpathmoveto{\pgfpoint{1cm}{0cm}}
  \pgfpathsine{\pgfpoint{-2cm}{-2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

\pgfpathcosine{⟨*vector*⟩}

This command appends a cosine curve in the interval $[0, \pi/2]$ to the current path. The curve is squeezed or stretched such that the curve starts at the current point and ends at the current point plus ⟨*vector*⟩. Using several sine and cosine operations in sequence allows you to produce a complete sine or cosine curve



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpoint{0cm}{0cm}}
  \pgfpathsine{\pgfpoint{1cm}{1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{-1cm}}
  \pgfpathsine{\pgfpoint{1cm}{-1cm}}
  \pgfpathcosine{\pgfpoint{1cm}{1cm}}
  \pgfsetfillcolor{yellow!80!black}
  \pgfusepath{fill,stroke}
\end{pgfpicture}
```

The command will apply coordinate transformations and update the bounding boxes.

102.11 Plot Path Operations

There exist several commands for appending plots to a path. These commands are available through the module `plot`. They are documented in Section 112.

102.12 Rounded Corners

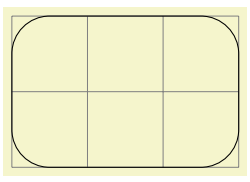
Normally, when you connect two straight line segments or when you connect two curves that end and start "at different angles", you get "sharp corners" between the lines or curves. In some cases it is desirable to produce "rounded corners" instead. Thus, the lines or curves should be shortened a bit and then connected by arcs.

PGF offers an easy way to achieve this effect, by calling the following two commands.

\pgfsetcornersarced{⟨*point*⟩}

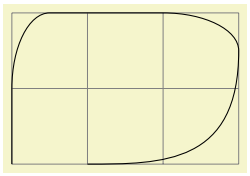
This command causes all subsequent corners to be replaced by little arcs. The effect of this command lasts till the end of the current TEX scope.

The ⟨*point*⟩ dictates how large the corner arc will be. Consider a corner made by two lines l and r and assume that the line l comes first on the path. The x-dimension of the ⟨*point*⟩ decides by how much the line l will be shortened, the y-dimension of ⟨*point*⟩ decides by how much the line r will be shortened. Then, the shortened lines are connected by an arc.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfsetcornersarced{\pgfpoint{5mm}{5mm}}
  \pgfpathrectanglecorners{\pgfpointorigin}{\pgfpoint{3cm}{2cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

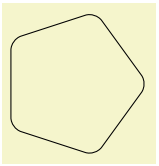


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);

  \pgfsetcornersarced{\pgfpoint{10mm}{5mm}}
  % 10mm entering,
  % 5mm leaving.
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{0cm}{2cm}}
  \pgfpathlineto{\pgfpoint{3cm}{2cm}}
  \pgfpathcurveto
    {\pgfpoint{3cm}{0cm}}
    {\pgfpoint{2cm}{0cm}}
    {\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

If the x- and y-coordinates of ⟨*point*⟩ are the same and the corner is a right angle, you will get a perfect quarter circle (well, not quite perfect, but perfect up to six decimals). When the angle is not 90°, you only get a fair approximation.

More or less "all" corners will be rounded, even the corner generated by a `\pgfpathclose` command. (The author is a bit proud of this feature.)



```
\begin{pgfpicture}
  \pgfsetcornersarced{\pgfpoint{4pt}{4pt}}
  \pgfpathmoveto{\pgfpointpolar{0}{1cm}}
  \pgfpathlineto{\pgfpointpolar{72}{1cm}}
  \pgfpathlineto{\pgfpointpolar{144}{1cm}}
  \pgfpathlineto{\pgfpointpolar{216}{1cm}}
  \pgfpathlineto{\pgfpointpolar{288}{1cm}}
  \pgfpathclose
  \pgfusepath{stroke}
\end{pgfpicture}
```

To return to normal (unrounded) corners, use `\pgfsetcornersarced{\pgfpointorigin}`.

Note that the rounding will produce strange and undesirable effects if the lines at the corners are too short. In this case the shortening may cause the lines to "suddenly extend over the other end" which is rarely desirable.

102.13 Internal Tracking of Bounding Boxes for Paths and Pictures

The path construction commands keep track of two bounding boxes: One for the current path, which is reset whenever the path is used and thereby flushed, and a bounding box for the current `{pgfpicture}`.

`\pgfresetboundingbox`

Resets the picture's bounding box. The picture will simply forget any previous bounding box updates and start collecting from scratch.

You can use this together with `\pgfusepath{use as bounding box}` to replace the bounding box by the one of a particular path (ignoring subsequent paths).

The bounding boxes are not accessible by "normal" macros. Rather, two sets of four dimension variables are used for this, all of which contain the letter `@`.

`\pgf@pathminx`

The minimum x-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxx`

The maximum x-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

`\pgf@pathminy`

The minimum y-coordinate "mentioned" in the current path. Initially, this is set to 16000pt.

`\pgf@pathmaxy`

The maximum y-coordinate "mentioned" in the current path. Initially, this is set to −16000pt.

\pgf@picminx

The minimum x-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

\pgf@picmaxx

The maximum x-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

\pgf@picminy

The minimum y-coordinate "mentioned" in the current picture. Initially, this is set to 16000pt.

\pgf@picmaxy

The maximum y-coordinate "mentioned" in the current picture. Initially, this is set to −16000pt.

Each time a path construction command is called, the above variables are (globally) updated. To facilitate this, you can use the following command:

\pgf@protocolsizes{⟨*x-dimension*⟩}{⟨*y-dimension*⟩}

Updates all of the above dimensions in such a way that the point specified by the two arguments is inside both bounding boxes. For the picture's bounding box this updating occurs only if `\ifpgf@relevantforpicturesize` is true, see below.

For the bounding box of the picture it is not always desirable that every path construction command affects this bounding box. For example, if you have just used a clip command, you do not want anything outside the clipping area to affect the bounding box. For this reason, there exists a special "TEX if" that (locally) decides whether updating should be applied to the picture's bounding box. Clipping will set this if to false, as will certain other commands.

\pgf@relevantforpicturesizefalse

Suppresses updating of the picture's bounding box.

\pgf@relevantforpicturesizetrue

Causes updating of the picture's bounding box.

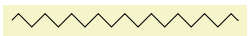
103 Decorations

```
\usepgfmodule{decorations}  % LaTeX and plain TeX when using pure PGF
\usepgfmodule[decorations]  % ConTeXt when using pure PGF
```

The commands for creating decorations are defined in this module, so you need to load this module to use decorations. This module is automatically loaded by the different decoration libraries.

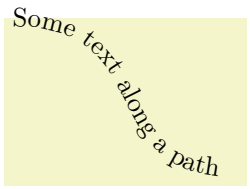
103.1 Overview

Decorations are a general way of creating graphics by "moving along" a path and, while doing so, either drawing something or constructing a new path. This could be as simple as extending a path with a "zigzagged" line…



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\tikz \draw decorate[decoration=zigzag] {(0,0) -- (3,0)};
```

…but could also be as complex as typesetting text along a path:




```
\usetikzlibrary {decorations,decorations.text}
\tikz \path decorate [decoration={text along path,
     text={Some text along a path}}]
   { (0,2) .. controls (2,2) and (1,0) .. (3,0) };
```

The workflow for using decorations is the following:

1. You define a decoration using the `\pgfdeclaredecoration` command. Different useful decorations are already declared in libraries like `decorations.shapes`.
2. You use normal path construction commands like `\pgfpathlineto` to construct a path. Let us call this path the *to-be-decorated* path.
3. You place the path construction commands inside the environment `{pgfdecoration}`. This environment takes the name of a previously declared decoration as a parameter. It will then start "walking along" the to-be-decorated path. As it does this, a special finite automaton called a *decoration automaton* produces new path commands as its output (or even other outputs). These outputs replace the to-be-decorated path; indeed, after the to-be-decorated path has been fully walked along it is thrown away, only the output of the automaton persists.

In the present section the process of how decoration automata work is explained first. Then the command(s) for declaring decoration automata and for using them are covered.

103.2 Decoration Automata

Decoration automata (and the closely related meta-decoration automata) are a general concept for creating graphics "along paths". For straight lines, this idea was first proposed by Till Tantau in an earlier version of PGF, the idea to extend this to arbitrary path was proposed and implemented by Mark Wibrow. Further versatility is provided by "meta-decorations". These are automata that decorate a path with decorations.

In the present subsection the different ideas underlying decoration automata are presented.

103.2.1 The Different Paths

In order to prevent confusion with different types of paths, such as those that are extended, those that are decorated and those that are created, the following conventions will be used:

- The *preexisting* path refers to the current path in existence before a decoration environment. (Possibly this path has been created by another decoration used earlier, but we will still call this path the preexisting path also in this case.)
- The *input* path refers to the to-be-decorated path that the decoration automaton moves along. The input path may consist of many line and curve input segments (for example, a circle or an ellipse consists of four curves). It is specified inside the decoration environment.

- The *output* path refers to the path that the decoration creates. Depending on the decoration, this path may or may not be empty (a decoration can also choose to use side-effects instead of producing an output path). The input path is always consumed by the decoration automaton, that is, it is no longer available in any way after the decoration automaton has finished.

The effect of a decoration environment is the following: The input path, which is specified inside the environment, is constructed and stored. This process does not alter the preexisting path in any way. Then the decoration automaton is started (as described later) and it produces an output path (possibly empty). Whenever part of the output path is produced, it is concatenated with the preexisting path. After the environment, the current path will equal the original preexisting path followed by the output path.

It is permissible that a decoration issues a `\pgfusepath` command. As usual, this causes the current path to be filled or stroked or some other action to be taken and the current path is set to the empty path. As described above, when the decoration automaton starts, the current path is the preexisting path and as the automaton progresses, the current path is constantly being extended by the output path. This means that first time a `\pgfusepath` command is used on a decoration, the preexisting path is part of the path this command operates on; in subsequent calls only the part of the output path constructed since the last `\pgfusepath` command will be used.

You can use this mechanism to stroke or fill different parts of the output path in different colors, line widths, fills and shades; all within the same decoration. Alternatively, a decoration can choose to produce no output path at all: the `text` decoration simply typesets text along a path.

103.2.2 Segments and States

The most common use of a decoration is to "repeat something along a path" (for example, the `zigzag` decoration repeats $\diagup\!\diagdown$ along a path). However, it not necessarily the case that only one thing is repeated: a decoration can consist of different parts, or *segments*, repeated in a particular order.

When you declare a decoration, you provide a description of how their different segments will be rendered. The description of each segment should be given in a way as if the "x-axis" of the segment is the tangent to the path at a particular point, and that point is the origin of the segment. Thus, for example, the segment of the `zigzag` decoration might be defined using the following code:

```
\pgfpathlineto{\pgfpoint{5pt}{5pt}}
\pgfpathlineto{\pgfpoint{15pt}{-5pt}}
\pgfpathlineto{\pgfpoint{20pt}{0pt}}
```

PGF will ensure that an appropriate coordinate transformation is in place when the segment is rendered such that the segment actually points in the right direction. Also, subsequent segments will be transformed such that they are "further along the path" toward the end of the path. All transformations are set up automatically.

Note that we did not use a `\pgfpathmoveto{\pgfpointorigin}` at the beginning of the segment code. Doing so would subdivide the path into numerous subpaths. Rather, we assume that the previous segment caused the current point to be at the origin.

The width of a segment can (and must) be specified explicitly. PGF will use this width to find out the start point of the next segment and the correct rotation. The width the you provide need not be the "real" width of the segment, which allows decoration segments to overlap or to be spaced far apart.

The `zigzag` decoration only has one segment that is repeated again and again. However, we might also like to have *different* segments and use rules to describe which segment should be used where. For example, we might have special segments at the start and at the end.

Decorations use a mechanism known in theoretical in computer science as *finite state automata* to describe which segment is used at a particular point. The idea is the following: For the first segment we start in a special *state* called the *initial state*. In this state, and also in all other states later, PGF first computes how much space is left on the input path. That is, PGF keeps track of the distance to the end of the input path. Attached to each state there is a set of rules of the following form: "If the remaining distance on the input path is less than x, switch to state q." PGF checks for each of these rules whether it applies and, if so, immediately switches to state q.

Only if none of the rules tell us to switch to another state, PGF will execute the state's code. This code will (typically) add a segment to the output path. In addition to the rules there is also a width parameter attached to each state. PGF then translates the coordinate system by this width and reduces the remaining distance on the input path. Then, PGF either stays in the current state or switches to another state, depending on yet another property attached of the state.

The whole process stops when a special state called `final` is reached. The segment of this state is immediately added to the output path (it is often empty, though) and the process ends.

103.3 Declaring Decorations

The following command is used to declare a decoration. Essentially, this command describes the decoration automaton.

`\pgfdeclaredecoration{`⟨*name*⟩`}{`⟨*initial state*⟩`}{`⟨*states*⟩`}`

This command declares a new decoration called ⟨*name*⟩. The ⟨*states*⟩ argument contains a description of the decoration automaton's states and the transitions between them. The ⟨*initial state*⟩ is the state in which the automaton starts.

When the automaton is later applied to an input path, it keeps track of a certain position on the input path. This current point will "travel along the path", each time being moved along by a certain distance. This will also work if the path is not a straight line. That is, it is permissible that the path curves are veers at a sharp angle. It is also permissible that while traveling along the input path, the current input segment ends and a new input segment starts. In this case, the remaining distance on the first input segment is subtracted from the ⟨*dimension*⟩ and then we travel along the second input segment for the remaining distance. This input segment may also end early, in which case we travel along the next input segment, and so on. Note that it cannot happen that we travel past the end of the input path since this would have caused an immediate switch to the `final` state.

Note that the computation of the path lengths has only a low accuracy because of TEX's small math capabilities. Do not expect high accuracy alignments when using decorations (unless the input path consists only of horizontal and vertical lines).

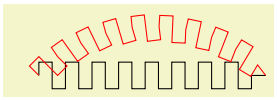
The ⟨*states*⟩ argument should consist of `\state` commands, one for each state of the decoration automaton. The `\state` command is defined only when the ⟨*states*⟩ argument is executed.

`\state{`⟨*name*⟩`}[`⟨*options*⟩`]{`⟨*code*⟩`}`

This command declares a new state inside the current decoration automaton. The state is named ⟨*name*⟩.

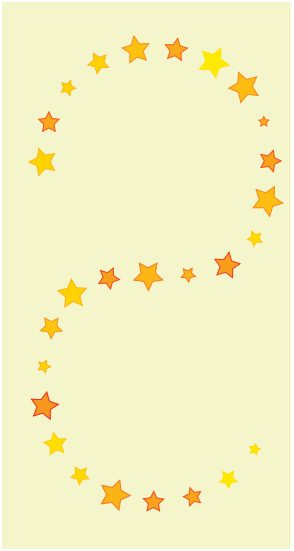
When the decoration automaton is in state ⟨*name*⟩, the following things happen:

1. The ⟨*options*⟩ are parsed. This may lead to a state switch, see below. When this happens, the following steps are not executed. The ⟨*options*⟩ are executed one after the other in the given order. If an option causes a state switch, the switch is immediate, even if later options might cause a different state switch.
2. The ⟨*code*⟩ is executed in a TEX-group with the current transformation matrix set up in such a way that the origin is on the input path at the current point (the point at the distance traveled up to now) and the coordinate system is rotated in such a way that the positive x-axis points in the direction of the tangent to the input path at the current point, while the positive y-axis points to the left of this tangent.
   As described earlier, the ⟨*code*⟩ can have two different effects: If it just contains path construction commands, the decoration will produce an output path, that is, it extends the preexisting path. Here is an example:



```
\usetikzlibrary {decorations}
\pgfdeclaredecoration{example}{initial}
{
  \state{initial}[width=10pt]
  {
    \pgfpathlineto{\pgfpoint{0pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{5pt}}
    \pgfpathlineto{\pgfpoint{5pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{-5pt}}
    \pgfpathlineto{\pgfpoint{10pt}{0pt}}
  }
  \state{final}
  {
    \pgfpathlineto{\pgfpointdecoratedpathlast}
  }
}
\tikz[decoration=example]
{
  \draw [decorate]     (0,0) -- (3,0);
  \draw [red,decorate] (0,0) to [out=45,in=135] (3,0);
}
```

Alternatively, the ⟨*code*⟩ can also contain the `\pgfusepath` command. This will use the path in the usual manner, where "the path" is the preexisting path plus a part of the output path for the first invocation and the different parts of the rest of the output path for the following invocation. Here is an example:



```
\usetikzlibrary {decorations,shapes.geometric}
\pgfdeclaredecoration{stars}{initial}{
  \state{initial}[width=15pt]
  {
    \pgfmathparse{round(rnd*100)}
    \pgfsetfillcolor{yellow!\pgfmathresult!orange}
    \pgfsetstrokecolor{yellow!\pgfmathresult!red}
    \pgfnode{star}{center}{}{}{\pgfusepath{stroke,fill}}
  }
  \state{final}
  {
    \pgfpathmoveto{\pgfpointdecoratedpathlast}
  }
}
\tikz\path[decorate, decoration=stars, star point ratio=2, star points=5,
           inner sep=0, minimum size=rnd*10pt+2pt]
  (0,0) .. controls (0,2)  and (3,2)  .. (3,0)
        .. controls (3,-3) and (0,0)  .. (0,-3)
        .. controls (0,-5) and (3,-5) .. (3,-3);
```

3. After the ⟨*code*⟩ has been executed (possibly more than once, if the `repeat state` option is used), the state switches to whatever state has been specified inside the ⟨*options*⟩ using the `next state` option. If no `next state` has been specified, the state stays the same.

The ⟨*options*⟩ are executed with the key path set to `/pgf/decoration automaton`. The following keys are defined:

`/pgf/decoration automaton/switch if less than=`⟨*dimension*⟩ `to` ⟨*new state*⟩ (no default)

When this key is encountered, PGF checks whether the remaining distance to the end of the input path is less than ⟨*dimension*⟩. If so, an immediate state switch to ⟨*new state*⟩ occurs.

`/pgf/decoration automaton/switch if input segment less than=`
⟨*dimension*⟩ `to` ⟨*new state*⟩ (no default)

When this key is encountered, PGF checks whether the remaining distance to the end of the current input segment of the input path is less than ⟨*dimension*⟩. If so, an immediate state switch to ⟨*new state*⟩ occurs.

`/pgf/decoration automaton/width=`⟨*dimension*⟩ (no default)

First, this option causes an immediate switch to the state `final` if the remaining distance

on the input path is less than ⟨*dimension*⟩. The effect is the same as if you had said `switch if less than=`⟨*dimension*⟩ `to final` just before the `width` option.

If no switch occurs, this option tells PGF the width of the segment. The current point will travel along the input path (as described earlier) by this distance.

`/pgf/decoration automaton/repeat state=`⟨*repetitions*⟩ (no default, initially `0`)

Tells PGF how long the automaton stays "normally" in the current state. This count is reset to ⟨*repetitions*⟩ each time one of the `switch if` keys causes a state switch. If no state switches occur, the ⟨*code*⟩ is executed and the repetition counter is decreased. Then, there is once more a chance of a state change caused by any of the ⟨*options*⟩. If no repetition occurs, the ⟨*code*⟩ is executed once more and the counter is decreased once more. When the counter reaches zero, the ⟨*code*⟩ is executed once more, but, then, a different state is entered, as specified by the `next state` option.

Note, that the maximum number of times the state will be executed is ⟨*repetitions*⟩ + 1.

`/pgf/decoration automaton/next state=`⟨*new state*⟩ (no default)

After the ⟨*code*⟩ for state has been executed for the last time, a state switch to ⟨*new state*⟩ is performed. If this option is not given, the next state is the same as the current state.

`/pgf/decoration automaton/if input segment is closepath=`⟨*options*⟩ (no default)

This key checks whether the current input segment is a closepath operation. If so, the ⟨*options*⟩ get executed; otherwise nothing happens. You can use this option to handle a closepath in some special way, for instance, switching to a new state in which `\pgfpathclose` is executed.

`/pgf/decoration automaton/auto end on length=`⟨*dimension*⟩ (no default)

This key is just included for convenience, it does nothing that cannot be achieved using the previous options. The effect is the following: If the remaining input path's length is at most ⟨*dimension*⟩, the decorated path is ended with a straight line to the end of the input path and, possibly, it is closed, namely if the input path ended with a closepath operation. Otherwise, it is checked whether the current input segment is a closepath segment and whether the remaining distance on the current input segment is at most ⟨*distance*⟩. If so, then a closepath operation is used to close the decorated path and the automaton continues with the next subpath, remaining in the current state.

In all other cases, nothing happens.

`/pgf/decoration automaton/auto corner on length=`⟨*dimension*⟩ (no default)

This key has the following effect: Firstly, in case the TEX-if `\ifpgfdecoratepathhascorners` is false, nothing happens. Otherwise, it is tested whether the remaining distance on the current input segment is at most ⟨*dimension*⟩. If so, a `lineto` operation is used to reach the end of this input segment and the automaton continues with the next input segment, but remains in the current state.

The main idea behind this option is to avoid having decoration segments "overshoot" past a corner.

You may sometimes wish to do computations outside the transformational TEX-group of the current segment, so that these results of these computations are available in the next state. For this, the following two options are useful:

`/pgf/decoration automaton/persistent precomputation=`⟨*precode*⟩ (no default)

If the ⟨*code*⟩ of the state is executed, the ⟨*precode*⟩ is executed first and it is executed outside the TEX-group of the ⟨*code*⟩. Note that when the ⟨*precode*⟩ is executed, the transformation matrix is not set up.

`/pgf/decoration automaton/persistent postcomputation=`⟨*postcode*⟩ (no default)

Works like the `persistent precomputation` option, only the ⟨*postcode*⟩ is executed after (and also outside) the TEX-group of the main ⟨*code*⟩.

There are a number of macros and dimensions which may be useful inside a decoration automaton. The following macros are available:

`\pgfdecoratedpathlength`

The length of the input path. If the input path consists of several input segments, this number is the sum of the lengths of the input segments.

`\pgfdecoratedinputsegmentlength`

The length of the current input segment of the input path. "Current input segment" refers to the input segment on which the current point lies.

`\pgfpointdecoratedpathlast`

The final point of the input path.

`\pgfpointdecoratedinputsegmentlast`

The final point of the current input segment of the input path.

`\pgfdecoratedangle`

The angle of the tangent to the decorated path at the *origin* of the current segment. The transformation matrix applied at the beginning of a state includes a rotation equivalent to this angle.

The following TEX dimension registers are also available inside the automaton:

`\pgfdecoratedremainingdistance`

The remaining distance on the input path.

`\pgfdecoratedcompleteddistance`

The completed distance on the input path.

`\pgfdecoratedinputsegmentremainingdistance`

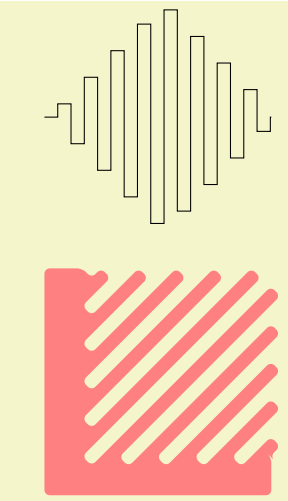
The remaining distance on the current input segment of the input path.

`\pgfdecoratedinputsegmentcompleteddistance`

The completed distance on the current input segment of the input path.

Further keys and macros are defined and used by the decoration libraries, see Section 50.

The following example shows how these options can be used:



```
\usetikzlibrary {decorations}
\pgfdeclaredecoration{complicated example decoration}{initial}
{
  \state{initial}[width=5pt,next state=up]
  { \pgfpathlineto{\pgfpoint{5pt}{0pt}} }

  \state{up}[width=5pt,next state=down]
  {
    \ifdim\pgfdecoratedremainingdistance>\pgfdecoratedcompleteddistance
      % Growing
      \pgfpathlineto{\pgfpoint{0pt}{\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      % Shrinking
      \pgfpathlineto{\pgfpoint{0pt}{\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \fi%
  }
  \state{down}[width=5pt,next state=up]
  {
    \ifdim\pgfdecoratedremainingdistance>\pgfdecoratedcompleteddistance
      % Growing
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfdecoratedcompleteddistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \else
      % Shrinking
      \pgfpathlineto{\pgfpoint{0pt}{-\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{-\pgfdecoratedremainingdistance}}
      \pgfpathlineto{\pgfpoint{5pt}{0pt}}
    \fi%
  }
  \state{final}
  {
    \pgfpathlineto{\pgfpointdecoratedpathlast}
  }
}
\begin{tikzpicture}[decoration=complicated example decoration]
  \draw decorate{ (0,0) -- (3,0)};
  \fill [red!50,rounded corners=2pt]
    decorate {(.5,-2) -- ++(2.5,-2.5)} -- (3,-5) -| (0,-2) -- cycle;
\end{tikzpicture}
```

103.3.1 Predefined Decorations

The three decorations `moveto`, `lineto`, and `curveto` are predefined and "always available". They are mostly useful in conjunction with meta-decorations. They are documented in Section 50 alongside the other decorations.

103.4 Using Decorations

Once a decoration has been declared, it can be used.

`\begin{pgfdecoration}{`⟨*decoration list*⟩`}`
⟨*environment contents*⟩
`\end{pgfdecoration}`

The ⟨*environment contents*⟩ should contain commands for creating an path. This path is the basis for the *input paths* for the decorations in the ⟨*decoration list*⟩. In detail, the following happens:

1. The preexisting unused path is saved.
2. The path commands specified in ⟨*environment contents*⟩ are executed and this resulting path is saved. The path is then divided into different *input paths* as follows: The format for each item in `{`⟨*decoration list*⟩`}` is

   `{`⟨*decoration*⟩`}{`⟨*length*⟩`}{`⟨*before code*⟩`}{`⟨*after code*⟩`}`

The ⟨*before code*⟩ and the ⟨*after code*⟩ are optional. The input path is divided into input paths as follows: The first input path consists of the first lines of the path specified in the ⟨*environment contents*⟩ until the ⟨*length*⟩ of the first element of the ⟨*decoration list*⟩ is reached. If this length is reached in the middle of a line, the line is broken up at this exact position. Then the second input path has the ⟨*length*⟩ of the second element in the ⟨*decoration list*⟩ and consists of the lines making up the following ⟨*length*⟩ part of the path in the ⟨*environment contents*⟩, and so on.

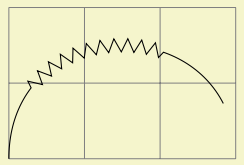
If the lengths in the ⟨*decoration list*⟩ do not add up to the total length of the path in the ⟨*environment contents*⟩, either some decorations are dropped (if their lengths add up to more than the length of the ⟨*environment contents*⟩) or the input path is not fully used (if their lengths add up to less).

3. The preexisting path is reinstalled.
4. The decoration automata move along the input paths, thus creating (and possibly using) the output paths. These output paths extend the current path (unless they are used).

Some important points should be noted regarding the use of this environment:

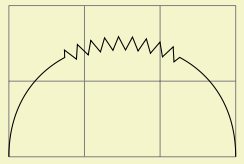
- If ⟨*environment contents*⟩ does not begin with `\pgfpathmoveto`, the last known point on the preexisting path is assumed as the starting point.
- All except the last of any sequence of consecutive move-to commands in ⟨*environment contents*⟩ are discarded.
- Any move-to commands at the end of ⟨*environment contents*⟩ are ignored.
- Any close-path commands on the input path are interpreted as straight lines. Internally, something a little more complicated is going on, however, a closed path on the input path has no effect on the output path, other than causing the automaton to travel in a straight line towards the location of the last move-to command on the input path.
- Although tangent computations for the input path work with the last point on the preexisting path, no automatic move-to operations are issued for the output path. If an output path starts with a line-to or curve-to when the existing path is empty, an appropriate move-to command should be inserted before the decoration starts.
- If a decoration uses its own path, the first time this happens the preexisting path is part of the path that is used at this point.

Before the automata start to "work on" their respective inputs paths, ⟨*before code*⟩ is executed. After the decoration automaton has finished, ⟨*after code*⟩ is executed.



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\begin{tikzpicture}[decoration={segment length=5pt}]
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{{curveto}{1cm},{zigzag}{2cm},{curveto}{1cm}}
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
\pgfusepath{stroke}
\end{tikzpicture}
```

When the lengths are evaluated, the dimension `\pgfdecoratedremainingdistance` holds the remaining distance on the entire decorated path, and `\pgfdecoratedpathlength` holds the total length of the path. Thus, it is possible to specify lengths like `\pgfdecoratedpathlength/3`.



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\begin{tikzpicture}[decoration={segment length=5pt}]
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{
      {curveto}{\pgfdecoratedpathlength/3},
      {zigzag}{\pgfdecoratedpathlength/3},
      {curveto}{\pgfdecoratedremainingdistance}
    }
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

When ⟨*before code*⟩ is executed, the following macro is useful:

\pgfpointdecoratedpathfirst

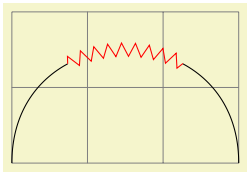
Returns the point corresponding to the start of the current input path.

When ⟨*after code*⟩ is executed, the following macro can be used:

\pgfpointdecoratedpathlast

Returns the point corresponding to the end of the current input path.

This means that if decorations do not use their own path, it is possible to do something with them and continue from the correct place.



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{
      {curveto}{\pgfdecoratedpathlength/3}
      {}
      {
        \pgfusepath{stroke}
      },
      {zigzag}{\pgfdecoratedpathlength/3}
      {
        \pgfpathmoveto{\pgfpointdecoratedpathfirst}
        \pgfdecorationsegmentlength=5pt
      }
      {
        \pgfsetstrokecolor{red}
        \pgfusepath{stroke}
        \pgfpathmoveto{\pgfpointdecoratedpathlast}
        \pgfsetstrokecolor{black}
      },
      {curveto}{\pgfdecoratedremainingdistance}
    }
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

After the `{decoration}` environment has finished, the following macros are available:

\pgfdecorateexistingpath

The preexisting path before the environment was entered.

\pgfdecoratedpath

The (total) input path (that is, the path created by the environment contents).

\pgfdecorationpath

The output path. If the path is used, this macro contains only the last unused part of the output path.

\pgfpointdecoratedpathlast

The final point of the input path.

\pgfpointdecorationpathlast

The final point of the output path.

The following style is executed each time a decoration is used. You may use it to set up default options for decorations.

/pgf/every decoration (style, initially empty)

This style is executed for every decoration.

\pgfdecoration{⟨*name*⟩}

⟨*environment contents*⟩
\endpgfdecoration

The plain TEX version of the {pgfdecoration} environment.

\startpgfdecoration{⟨*name*⟩}
⟨*environment contents*⟩
\stoppgfdecoration

The ConTEXt version of the {pgfdecoration} environment.

For convenience, the following macros provide a "shorthand" for decorations (internally, they all use the {pgfdecoration} environment).

\pgfdecoratepath{⟨*name*⟩}{⟨*path commands*⟩}

Decorate the path described by ⟨*path commands*⟩ with the decoration ⟨*name*⟩. This is equivalent to

```
\pgfdecoration{{name}{\pgfdecoratedpathlength}
              {\pgfdecoratebeforecode}{\pgfdecorateaftercode}}
  // the path commands.
\endpgfdecoration
```

\pgfdecoratecurrentpath{⟨*name*⟩}

Decorate the preexisting path with the decoration ⟨*name*⟩.

Both the above commands use the current definitions of the following macros:

\pgfdecoratebeforecode

Code executed as ⟨*before code*⟩, see the description of \pgfdecoration.

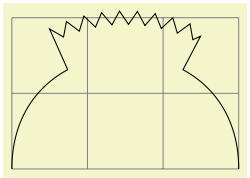
\pgfdecorateaftercode

Code executed as ⟨*after code*⟩, see the description of \pgfdecoration.

It may sometimes be useful to add an additional transformation for each segment of a decoration. The following command allows you to define such a "last minute transformation".

\pgfsetdecorationsegmenttransformation{⟨*code*⟩}

The ⟨*code*⟩ will be executed at the very beginning of each segment. Note when applying multiple decorations, this will be reset between decorations, so it needs to be specified for each segment.



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\begin{tikzpicture}
  \draw [help lines] grid (3,2);
  \begin{pgfdecoration}{
      {curveto}{\pgfdecoratedpathlength/3},
      {zigzag}{\pgfdecoratedpathlength/3}
      {
        \pgfdecorationsegmentlength=5pt
        \pgfsetdecorationsegmenttransformation{\pgftransformyshift{.5cm}}
      },
      {curveto}{\pgfdecoratedremainingdistance}
    }
    \pgfpathmoveto{\pgfpointorigin}
    \pgfpathcurveto
      {\pgfpoint{0cm}{2cm}}{\pgfpoint{3cm}{2cm}}{\pgfpoint{3cm}{0cm}}
  \end{pgfdecoration}
  \pgfusepath{stroke}
\end{tikzpicture}
```

103.5 Meta-Decorations

A meta-decoration provides an alternative way to decorate a path with multiple decorations. It is, in essence, an automaton that decorates an input path with decoration automatons. In general, however, the end effect is still that a path is decorated with other paths, and the input path should be thought of as being divided into sub-input-paths, each with their own decoration. Like ordinary decorations, a meta-decoration must be declared before it can be used.

103.5.1 Declaring Meta-Decorations

\pgfdeclaremetadecorate{⟨*name*⟩}{⟨*initial state*⟩}{⟨*states*⟩}

This command declares a new meta-decoration called ⟨*name*⟩. The ⟨*states*⟩ argument contains a description of the meta-decoration automaton's states and the transitions between them. The ⟨*initial state*⟩ is the state in which the automaton starts.

The `\state` command is similar to the one found in decoration declarations, and takes the same form:

\state{⟨*name*⟩}[⟨*options*⟩]{⟨*code*⟩}

Declares the state ⟨*name*⟩ inside the current meta-decoration automaton. Unlike decorations, states in meta-decorations are not executed within a group, which makes the persistent computation options superfluous. Consider using an initial state with `width=0pt` to do precalculations that could speed the execution of the meta-decoration.

The ⟨*options*⟩ are executed with the key path set to `/pgf/meta-decorations automaton/`, and the following keys are defined for this path:

/pgf/meta-decoration automaton/switch if less than=⟨*dimension*⟩ to ⟨*new state*⟩ (no default)

This causes PGF to check whether the remaining distance to the end of the input path is less than ⟨*dimension*⟩, and, if so, to immediately switch to the state ⟨*new state*⟩. When this key is evaluated, the macro `\pgfmetadecoratedpathlength` will be defined as the total length of the decoration path, allowing for values such as `\pgfmetadecoratedpathlength/8`.

/pgf/meta-decoration automaton/width=⟨*dimension*⟩ (no default)

As always, this option will cause an immediate switch to the state `final` if the remaining distance on the input path is less than ⟨*dimension*⟩.

Otherwise, this option tells PGF the width of the "meta-segment", that is, the length of the sub-input-path which the decoration automaton specified in ⟨*code*⟩ will decorate.

/pgf/meta-decoration automaton/next state=⟨*new state*⟩ (no default)

After the code for a state has been executed, a state switch to ⟨*new state*⟩ is performed. If this option is not given, the next state is the same as the current state.

The code in ⟨*code*⟩ is quite different from the code in a decoration state. In almost all cases only the following three macros will be required:

\decoration{⟨*name*⟩}

This sets the decoration for the current state to ⟨*name*⟩. If this command is omitted, the `moveto` decoration will be used.

\beforedecoration{⟨*before code*⟩}

Defines ⟨*before code*⟩ as (typically) PGF commands to be executed before the decoration is applied to the current segment. This command can be omitted. If you wish to set up some decoration specific parameters such as segment length, or segment amplitude, then they can be set in ⟨*before code*⟩.

\afterdecoration{⟨*after code*⟩}

Defines ⟨*after code*⟩ as commands to be executed after the decoration has been applied to the current segment. This command can be omitted.

There are some macros that may be useful when creating meta-decorations (note that they are all macros):

\pgfpointmetadecoratedpathfirst

When the ⟨*before code*⟩ is executed, this macro stores the first point on the current sub-input-path.

\pgfpointmetadecoratedpathlast

When the ⟨*after code*⟩ is executed, this macro stores the last point on the current sub-input-path.

\pgfmetadecoratedpathlength

The entire length of the entire input path.

\pgfmetadecoratedcompleteddistance

The completed distance on the entire input path.

\pgfmetadecoratedremainingdistance

The remaining distance on the entire input path.

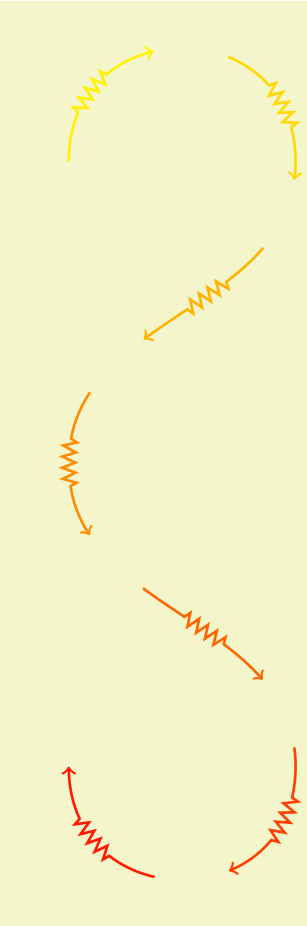
\pgfmetadecoratedinputsegmentcompleteddistance

The completed distance on the current input segment of the entire input path.

\pgfmetadecoratedinputsegmentremainingdistance

The remaining distance on the current input segment of the entire input path.

Here is a complete example of a meta-decoration:



```
\usetikzlibrary {decorations,decorations.pathmorphing}
\pgfdeclaremetadecoration{arrows}{initial}{
  \state{initial}[width=0pt, next state=arrow]
  {
    \pgfmathdivide{100}{\pgfmetadecoratedpathlength}
    \let\factor\pgfmathresult
    \pgfsetlinewidth{1pt}
    \pgfset{/pgf/decoration/segment length=4pt}
  }
  \state{arrow}[
    switch if less than=\pgfmetadecorationsegmentlength to final,
    width=\pgfmetadecorationsegmentlength/3,
    next state=zigzag]
  {
    \decoration{curveto}
    \beforedecoration
    {
      \pgfmathparse{\pgfmetadecoratedcompleteddistance*\factor}
      \pgfsetcolor{red!\pgfmathresult!yellow}
      \pgfpathmoveto{\pgfpointmetadecoratedpathfirst}
    }
  }
  \state{zigzag}[width=\pgfmetadecorationsegmentlength/3, next state=end arrow]
  {
    \decoration{zigzag}
  }
  \state{end arrow}[width=\pgfmetadecorationsegmentlength/3, next state=move]
  {
    \decoration{curveto}
    \beforedecoration{\pgfpathmoveto{\pgfpointmetadecoratedpathfirst}}
    \afterdecoration
    {
      \pgfsetarrowsend{to}
      \pgfusepath{stroke}
    }
  }
  \state{move}[width=\pgfmetadecorationsegmentlength/2, next state=arrow]{}
  \state{final}{}
}

\tikz\draw[decorate,decoration={arrows,meta-segment length=2cm}]
  (0,0) .. controls (0,2)   and (3,2)   .. (3,0)
        .. controls (3,-2)  and (0,-2)  .. (0,-4)
        .. controls (0,-6)  and (3,-6)  .. (3,-8)
        .. controls (3,-10) and (0,-10) .. (0,-8);
```

103.5.2 Predefined Meta-decorations

There are no predefined meta-decorations loaded with PGF.

103.5.3 Using Meta-Decorations

Using meta-decorations is “simpler” than using decorations, because you can only use one meta-decoration per path.

```
\begin{pgfmetadecoration}{⟨name⟩}
  ⟨environment contents⟩
\end{pgfmetadecoration}
```

This environment decorates the input path described in ⟨*environment contents*⟩, with the meta-decoration ⟨*name*⟩.

```
\pgfmetadecoration{⟨name⟩}
  ⟨environment contents⟩
\endpgfmetadecoration
```

The plain TEX version of the environment.

```
\startpgfmetadecoration{⟨name⟩}
  ⟨environment contents⟩
\stoppgfmetadecoration
```

The ConTEXt version of the environment.

104 Using Paths

104.1 Overview

Once a path has been constructed, it can be *used* in different ways. For example, you can draw the path or fill it or use it for clipping.

Numerous graph parameters influence how a path will be rendered. For example, when you draw a path, the line width is important as well as the dashing pattern. The options that govern how paths are rendered can all be set with commands starting with `\pgfset`. *All options that influence how a path is rendered always influence the complete path.* Thus, it is not possible to draw part of a path using, say, a red color and drawing another part using a green color. To achieve such an effect, you must use two paths.

In detail, paths can be used in the following ways:

1. You can *stroke* (also known as *draw*) a path.
2. You can add *arrow tips* to a path.
3. You can *fill* a path with a uniform color.
4. You can *clip* subsequent renderings against the path.
5. You can *shade* a path.
6. You can *use the path as bounding box* for the whole picture.

You can also perform any combination of the above, though it makes no sense to fill and shade a path at the same time.

To perform (a combination of) the first four actions, you can use the following command:

`\pgfusepath{`⟨*actions*⟩`}`

Applies the given ⟨*actions*⟩ to the current path. Afterwards, the current path is (globally) empty. The following actions are possible:

- `fill` fills the path. See Section 104.4 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `stroke` strokes the path. See Section 104.2 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

- `draw` has the same effect as `stroke`.
- `clip` clips all subsequent drawings against the path. Always suppresses arrow tips. See Section 104.5 for further details.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke,clip}
  \pgfpathcircle{\pgfpoint{1cm}{1cm}}{0.5cm}
  \pgfusepath{fill}
\end{pgfpicture}
```

- `discard` discards the path, that is, it is not used at all. Giving this option (alone) has the same effect as giving an empty options list.

When more than one of the first three actions are given, they are applied in the above ordering, regardless of their ordering in ⟨*actions*⟩. Thus, `{stroke,fill}` and `{fill,stroke}` have the same effect.

To shade a path, use the `\pgfshadepath` command, which is explained in Section 114.

104.2 Stroking a Path

When you use `\pgfusepath{stroke}` to stroke a path, several graphic parameters influence how the path is drawn. The commands for setting these parameters are explained in the following.

Note that all graphic parameters apply to the path as a whole, never only to a part of it.

All graphic parameters are local to the current `{pgfscope}`, but they persists past TEX groups, *except* for the interior rule (even-odd or nonzero) and the arrow tip kinds. The latter graphic parameters only persist till the end of the current TEX group, but this may change in the future, so do not count on this.

104.2.1 Graphic Parameter: Line Width

`\pgfsetlinewidth{`⟨*line width*⟩`}`

This command sets the line width for subsequent strokes (in the current `pgfscope`). The line width is given as a normal TEX dimension like `0.4pt` or `1mm`.



```
\begin{pgfpicture}
  \pgfsetlinewidth{1mm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetlinewidth{2\pgflinewidth} % double in size
  \pgfpathmoveto{\pgfpoint{0mm}{5mm}}
  \pgfpathlineto{\pgfpoint{2cm}{5mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgflinewidth`

You can access the current line width via the TEX dimension `\pgflinewidth`. It will be set to the correct line width, that is, even when a TEX group closed, the value will be correct since it is set globally, but when a `{pgfscope}` closes, the value is set to the correct value it had before the scope.

104.2.2 Graphic Parameter: Caps and Joins

`\pgfsetbuttcap`

Sets the line cap to a butt cap. See Section 15.3.1 for an explanation of what this is.

`\pgfsetroundcap`

Sets the line cap to a round cap. See again Section 15.3.1.

`\pgfsetrectcap`

Sets the line cap to a square cap. See again Section 15.3.1.

`\pgfsetroundjoin`

Sets the line join to a round join. See again Section 15.3.1.

`\pgfsetbeveljoin`

Sets the line join to a bevel join. See again Section 15.3.1.

`\pgfsetmiterjoin`

Sets the line join to a miter join. See again Section 15.3.1.

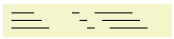
`\pgfsetmiterlimit{`⟨*miter limit factor*⟩`}`

Sets the miter limit to ⟨*miter limit factor*⟩. See again Section 15.3.1.

104.2.3 Graphic Parameter: Dashing

`\pgfsetdash{`⟨*list of even length of dimensions*⟩`}{`⟨*phase*⟩`}`

Sets the dashing of a line. The first entry in the list specifies the length of the first solid part of the list. The second entry specifies the length of the following gap. Then comes the length of the second solid part, following by the length of the second gap, and so on. The ⟨*phase*⟩ specifies where the first solid part starts relative to the beginning of the line.



```
\begin{pgfpicture}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0cm}
  \pgfpathmoveto{\pgfpoint{0mm}{0mm}}
  \pgfpathlineto{\pgfpoint{2cm}{0mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.1cm}
  \pgfpathmoveto{\pgfpoint{0mm}{1mm}}
  \pgfpathlineto{\pgfpoint{2cm}{1mm}}
  \pgfusepath{stroke}
  \pgfsetdash{{0.5cm}{0.5cm}{0.1cm}{0.2cm}}{0.2cm}
  \pgfpathmoveto{\pgfpoint{0mm}{2mm}}
  \pgfpathlineto{\pgfpoint{2cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

Use `\pgfsetdash{}{0pt}` to get a solid dashing.

104.2.4 Graphic Parameter: Stroke Color

`\pgfsetstrokecolor{`⟨*color*⟩`}`

Sets the color used for stroking lines to ⟨*color*⟩, where ⟨*color*⟩ is a LaTeX color like `red` or `black!20!red`. Unlike the `\color` command, the effect of this command lasts till the end of the current `{pgfscope}` and not till the end of the current TeX group.

The color used for stroking may be different from the color used for filling. However, a `\color` command will always "immediately override" any special settings for the stroke and fill colors.

In plain TeX, this command will also work, but the problem of *defining* a color arises. After all, plain TeX does not provide LaTeX colors. For this reason, PGF implements a minimalistic "emulation" of the `\definecolor`, `\colorlet`, and `\color` commands. Only gray-scale and rgb colors are supported. For most cases this turns out to be enough.



```
\begin{pgfpicture}
  \pgfsetlinewidth{1pt}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \pgfsetstrokecolor{black}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
  \color{red}
  \pgfpathcircle{\pgfpoint{2cm}{0cm}}{3mm} \pgfusepath{fill,stroke}
\end{pgfpicture}
```

`\pgfsetcolor{`⟨*color*⟩`}`

Sets both the stroke and fill color. The difference to the normal `\color` command is that the effect lasts till the end of the current `{pgfscope}`, not only till the end of the current TeX group.

104.2.5 Graphic Parameter: Stroke Opacity

You can set the stroke opacity using `\pgfsetstrokeopacity`. This command is described in Section 115.

104.2.6 Inner Lines

When a path is stroked, it is possible to request that it is stroked twice, the second time with a different line width and a different color. This is a useful effect for creating "double" lines, for instance by setting the line width to 2pt and stroking a black line and then setting the inner line width to 1pt and stroking a white line on the same path as the original path. This results in what looks like two lines, each of thickness 0.5pt, spaced 1pt apart.

You may wonder why there is direct support for this "double stroking" in the basic layer. After all, this effect is easy to achieve "by hand". The main reason is that arrow tips must be treated in a special manner when such "double lines" are present. First, the order of actions is important: First, the (thick) main line should be stroked, then the (thin) inner line, and only then should the arrow tip be drawn. Second, the way an arrow tip looks typically depends strongly on the width of the inner line, so the arrow tip code, which is part of the basic layer, needs access to the inner line thickness.

Two commands are used to set the inner line width and color.

\pgfsetinnerlinewidth{⟨*dimension*⟩}

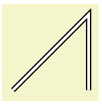
This command sets the width of the inner line. Whenever a path is stroked (and only then), it will be stroked normally and, afterward, it is stroked once more with the color set to the inner line color and the line width set to ⟨*dimension*⟩.

In case arrow tips are added to a path, the path is first stroked normally, then the inner line is stroked, and then the arrow tip is added. In case the main path is shortened because of the added arrow tip, this shortened path is double stroked, not the original path (which is exactly what you want).

When the inner line width is set to 0pt, which is the default, no inner line is stroked at all (not even a line of width 0pt). So, in order to "switch off" double stroking, set ⟨*dimension*⟩ to `0pt`.

The setting of the inner line width is local to the current TEX group and *not* to the current PGF scope.

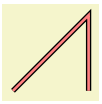
Note that inner lines will *not* be drawn for paths that are also used for clipping. However, this may change in the future, so you should not depend on this.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfsetlinewidth{2pt}
  \pgfsetinnerlinewidth{1pt}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetinnerstrokecolor{⟨*color*⟩}

This command sets the ⟨*color*⟩ that is to be used when the inner line is stroked. The effect of this command is also local to the current TEX group.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{1cm}}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfsetlinewidth{2pt}
  \pgfsetinnerlinewidth{1pt}
  \pgfsetinnerstrokecolor{red!50}
  \pgfusepath{stroke}
\end{pgfpicture}
```

104.3 Arrow Tips on a Path

After a path has been drawn, PGF can add arrow tips at the ends, depending on how the `tips` key is set, on whether `stroke` or `clip` are used and on whether the path contains closed subpaths. The exact rules when arrow tips are added are explained in Section 16.2.

\pgfsetarrowsstart{⟨*start arrow tip specification*⟩}

Sets the arrow tip kind used at the start of a (possibly curved) path. The syntax of the ⟨*start arrow specification*⟩ is detailed in Section 16.4.

To "clear" the start arrow, say `\pgfsetarrowsstart{}`.



```
\usepgflibrary {arrows.meta}
\begin{pgfpicture}
  \pgfsetarrowsstart{Latex[length=10pt]}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
  \pgfsetarrowsstart{Computer Modern Rightarrow}
  \pgfpathmoveto{\pgfpoint{0cm}{2mm}}
  \pgfpathlineto{\pgfpoint{1cm}{2mm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The effect of this command persists only till the end of the current TEX scope.

\pgfsetarrowsend{⟨*end arrow tip specification*⟩}

Sets the arrow tip kind used at the end of a path.



```
\usepgflibrary {arrows.meta}
\begin{pgfpicture}
  \pgfsetarrowsstart{Latex[length=10pt]}
  \pgfsetarrowsend{Computer Modern Rightarrow}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetarrows{⟨*argument*⟩}

The ⟨*argument*⟩ can be of the form ⟨*start arrow tip specification*⟩-⟨*end arrow tip specification*⟩. In this case, both the start and the end arrow specification are set:



```
\usepgflibrary {arrows.meta}
\begin{pgfpicture}
  \pgfsetarrows{Latex[length=10pt]->>}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{1cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

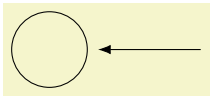
Alternatively, ⟨*argument*⟩ can be of the form `[`⟨*arrow keys*⟩`]`. In this case, the ⟨*arrow keys*⟩ will be set for all arrow tips in the current scope, see Section 16.4.5.

\pgfsetshortenstart{⟨*dimension*⟩}

This command will shortened the start of every stroked path by the given dimension. This shortening is done in addition to automatic shortening done by a start arrow, but it can be used even if no start arrow is given.

It is usually better to use the `sep` key with arrow tips.

This command is useful if you wish arrows or lines to "stop shortly before" a given point.



```
\usepgflibrary {arrows.meta}
\begin{pgfpicture}
  \pgfpathcircle{\pgfpointorigin}{5mm}
  \pgfusepath{stroke}
  \pgfsetarrows{Latex-}
  \pgfsetshortenstart{4pt}
  \pgfpathmoveto{\pgfpoint{5mm}{0cm}} % would be on the circle
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfsetshortenend{⟨*dimension*⟩}

Works like `\pgfsetshortenstart`.

104.4 Filling a Path

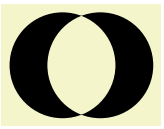
Filling a path means coloring every interior point of the path with the current fill color. It is not always obvious whether a point is "inside" a path when the path is self-intersecting and/or consists or multiple parts. In this case either the nonzero winding number rule or the even-odd crossing number rule is used to decide which points lie "inside". These rules are explained in Section 15.5.

104.4.1 Graphic Parameter: Interior Rule

You can set which rule is used using the following commands:

\pgfseteorule

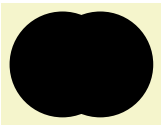
Dictates that the even-odd rule is used in subsequent fillings in the current *TEX scope*. Thus, for once, the effect of this command does not persist past the current TEX scope.



```
\begin{pgfpicture}
  \pgfseteorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

\pgfsetnonzerorule

Dictates that the nonzero winding number rule is used in subsequent fillings in the current TEX scope. This is the default.



```
\begin{pgfpicture}
  \pgfsetnonzerorule
  \pgfpathcircle{\pgfpoint{0mm}{0cm}}{7mm}
  \pgfpathcircle{\pgfpoint{5mm}{0cm}}{7mm}
  \pgfusepath{fill}
\end{pgfpicture}
```

104.4.2 Graphic Parameter: Filling Color

\pgfsetfillcolor{⟨*color*⟩}

Sets the color used for filling paths to ⟨*color*⟩. Like the stroke color, the effect lasts only till the next use of `\color`.

104.4.3 Graphic Parameter: Fill Opacity

You can set the fill opacity using `\pgfsetfillopacity`. This command is described in Section 115.

104.5 Clipping a Path

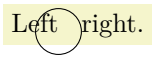
When you add the `clip` option, the current path is used for clipping subsequent drawings. The same rule as for filling is used to decide whether a point is inside or outside the path, that is, either the even-odd rule or the nonzero rule.

Clipping never enlarges the clipping area. Thus, when you clip against a certain path and then clip again against another path, you clip against the intersection of both.

The only way to enlarge the clipping path is to end the `{pgfscope}` in which the clipping was done. At the end of a `{pgfscope}` the clipping path that was in force at the beginning of the scope is reinstalled.

104.6 Using a Path as a Bounding Box

When you add the `use as bounding box` option, the bounding box of the picture will be enlarged such that the path in encompassed, but any *subsequent* paths of the current TEX scope will not have any effect on the size of the bounding box. Typically, you use this command at the very beginning of a `{pgfpicture}` environment. Alternatively, you can use `\pgfresetboundingbox`, followed by `\pgfusepath{use as bounding box}` to overrule the picture's bounding box completely.




```
Left
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2ex}{1ex}}
  \pgfusepath{use as bounding box} % draws nothing

  \pgfpathcircle{\pgfpointorigin}{2ex}
  \pgfusepath{stroke}
\end{pgfpicture}
right.
```

105 Defining New Arrow Tip Kinds

105.1 Overview

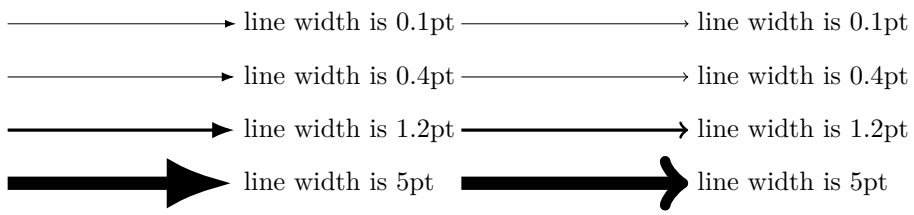
In present section we have a look at how you can define new arrow tips for use in PGF. The low-level commands for selecting which arrow tips are to be used have already been described in Section 104.3, the general syntax rules for using arrows are detailed in Section 16. Although Section 16 describes the use of arrows in Ti*k*Z, in reality, Ti*k*Z itself does not actually do anything about arrow tips; all of the functionality is implemented on the PGF level in the commands described in Section 16. Indeed, even the `/.tip` key handler described in Section 16 is actually implemented on the PGF layer.

What has *not* yet been covered is how you can actually define a complete new arrow tip. In PGF, arrows are "meta-arrows" in the same way that fonts in TEX are "meta-fonts". When a meta-arrow is resized, it is not simply scaled, but a possibly complicated transformation is applied to the size.

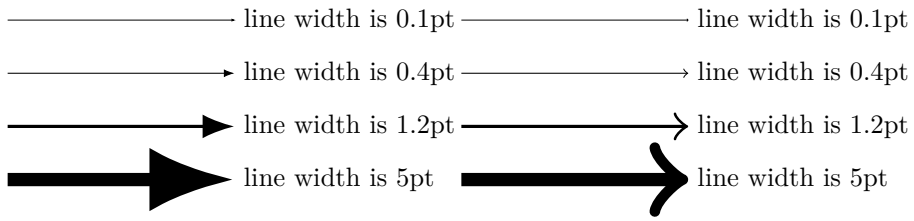
A meta-font is not one particular font at a specific size with a specific stroke width (and with a large number of other parameters being fixed). Rather, it is a "blueprint" (actually, more like a program) for generating such a font at a particular size and width. This allows the designer of a meta-font to make sure that, say, the font is somewhat thicker and wider at very small sizes. To appreciate the difference: Compare the following texts: "Berlin" and "Berlin". The first is a "normal" text, the second is the tiny version scaled by a factor of two. Obviously, the first look better. Now, compare "Berlin" and "Berlin". This time, the normal text was scaled down, while the second text is a "normal" tiny text. The second text is easier to read.

PGF's meta-arrows work in a similar fashion: The shape of an arrow tip can vary according to a great number of parameters, the line width of the arrow tip being one of them. Thus, an arrow tip drawn at a line width of 5pt will typically *not* be five times as large as an arrow tip of line width 1pt. Instead, the size of the arrow will get bigger only slowly as the line width increases.

To appreciate the difference, here are the `Latex` and `Classical TikZ Rightarrow` arrows, as drawn by PGF at four different sizes:



Here, by comparison, are the same arrows when they are simply "resized":

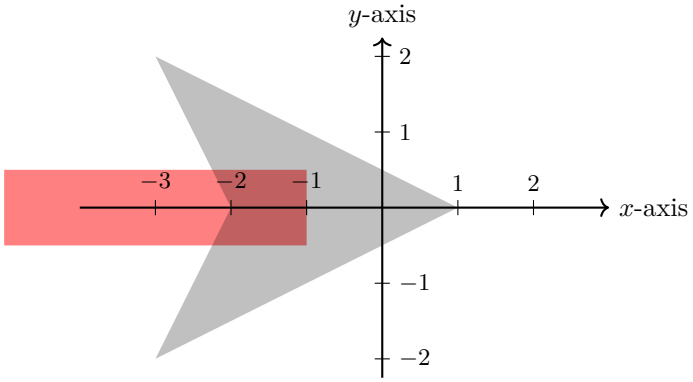


As can be seen, simple scaling produces arrow tips that are way too large at larger sizes and way too small at smaller sizes.

In addition to the line width, other options may also influence the appearance of an arrow tip. In particular, the width of the inner line (the line used to create the effect of a double line) influences arrow tips as well as other options that are specific to the arrow tip.

105.2 Terminology

Before we have a look at the exact commands used for defining arrow tips, we need to fix some terminology. Consider the following drawing of an arrow tip where the arrow tip is drawn transparently so that we can



see what is "happening behind it":

I have also added a coordinate system. The code for drawing an arrow tip always draws it in the way shown above: Pointing right along the x-axis.

We will use the following terminology:

- The point where tip of the arrow ends is called the *tip end*. It is at $(1, 0)$ in our example and we always assume it to lie on the x-axis, so we just treat it as a distance, 1 in this case. This is the position where the original path was supposed to end (so if the arrow tip had not been added to the red path, it would have ended here).
- The *back end* of the arrow is where a vertical line just to the left of the arrow intersects the x-axis. In our case, this is the point $(-3, 0)$ and again we treat it as a distance, -3 in this case.
- The *line end* is the position where the path now ends. This should be a position inside the arrow head that gets "covered" by the path. Note that a path may have a round or a rect head and should still be covered. Clearly, necessary shortening of the path will be the difference between the tip end and the line end.
- The *visual back end* is the position where the path and the the arrow head "meet last" on the path. In our case, because of the inset, the visual back end is not the same as the back end: The arrow ends "visually" at $(-2, 0)$. The difference between the back end and the visual back end is important when the arrow tip is flexed, see Section 16.3.8 for an explanation of flexing.
- There is also a *visual tip end*, the counterpart of the visual back end for the front. In our case, the visual tip end and the tip end obviously coincide, but if we were to reverse the arrow tip, the visual tip end would be different from the tip end (while the visual back end would then coincide with the new back end).
- There are four points that make up the *convex hull* of the arrow tip: $(1, 0)$, $(-3, 2)$, and $(-3, -2)$.

  Normally, PGF automatically keeps track of a bounding box of everything you draw. However, since arrow tips are drawn so often, PGF caches the code needing for drawing arrow tips internally and because of this cache it cannot determine the size of the arrow tip just based on the drawing commands used for drawing the tip. Instead, a convex hull of the arrow tip must be explicitly provided in the definition.

When you design a new arrow tip, all of the above parameters must be defined.

105.3 Caching and Rendering of Arrows

As a last preparation for the description of the commands for declaring arrows, it is important to understand the exact process by which PGF draws arrows.

1. First, you have to define an arrow tip kind using `\pgfdeclarearrow{name=foo,...}`. This will tell PGF that `foo` is now the name of an arrow tip. In particular, the parser for arrow tip specifications will now treat `foo` as the name of an arrow tip and will not try to consider `f`, `o`, and `o` as the names of single-char shorthands.

   Other than storing the definitions in the declaration internally, this command has little other effect. In particular, no drawing or other processing takes place.

2. Now assume that at some point the arrow tip `foo` is actually used. In this case, certain options may have been set, for instance the user may have requested the arrow tip `foo[length=5pt,open]`. What happens next depends on whether it is the first time the arrow tip `foo` is used with *these exact options* or not.

3. Assume that is the first time `foo` is requested at a length of 5pt and in an "open" version. PGF now retrieves the definition of the arrow tip kind that it stored in the first step and executes the so-called *setup* code. When this code is executed, all the options will be in force (for instance, `\pgfarrowlength` will equal `5pt` in our case). The job of the setup code is two-fold: First, it needs to compute all of the parameters listed in Section 105.2, that is, it has to compute where the tip end will lie in the arrow tip's coordinate system *at the particular size of 5pt*, where the back end will be, where the convex hull points lie, and so on. Second, the setup code should precompute values that will be important for constructing the path of the arrow. In our example, there is little to do in this regard, but for more complicated arrows, all time-consuming preparations are done now.

   It is *not* the job of the setup to actually draw the arrow tip, only to "prepare" this as much as possible.

   The setup code will always be executed only once for each arrow tip kind for a given set of options. Thus, when a user uses `foo[length=5pt,open]` once more later anywhere in the document, the setup code will not be executed again.

4. The next thing that happens is that we have a look at the *drawing code* stored in the `code` field of the arrow. In our example, the drawing code would consist of creating a filled path with four straight segments.

   In most cases, what happens now is that the drawing code is executed in a special sandbox in which the low-level driver commands that do the actual drawing are intercepted and stored away in a so-called *cache*. Once such a cache has been created, its contents will be reused whenever `foo[length=5pt,open]` is requested by a user and just like the setup code, the drawing code will not be executed again.

   There are, however, two cases in which the drawing code gets executed each time the arrow is used: First, an arrow tip kind can specify that this should always happen by saying `cache=false` in its definition. This is necessary if the drawing code contains low-level drawing commands that cannot be intercepted such as a use of `\pgftext` for arrow tips that "contain text". Second, when the `bend` option is used, the same arrow tip will look different each time it is used, namely in dependence on the exact curvature of the path to which it is added.

   Because the drawing code may be executed several times, while the setup code may not, we must find a way to "communicate" the values computed by the setup code to the drawing code. This is done by explicitly calling `\pgfarrowssave` inside the setup code. Whatever is "saved" in this way is restored each time before the drawing code is executed.

As can be seen, the process is a bit involved, but it leads to a reasonably fast arrow tip management.

105.4 Declaring an Arrow Tip Kind

`\pgfdeclarearrow{`⟨*config*⟩`}`

This command is both used to define a new arrow tip kind and to to declare a so-called shorthand. We have a look at the case that a complete new arrow tip kind is created and then have a look how the command can be used to create shorthands.

Defining a Complete New Arrow Tip Kind. The ⟨*config*⟩ is a key–value list in which different keys are used to setup the to-be defined arrow. The following keys can be given:

- `name=`⟨*name*⟩ or `name=`⟨*start name*⟩`-`⟨*end name*⟩
  This defines the name of the arrow tip. It is legal to define an arrow tip a second time, in this case the previous definition will be overwritten in the current TEX scope. It is customary to use a name with an uppercase first letter for a "complete" arrow tip kind. Short names and lower case names should be used for shorthands that change their meaning inside a document, while arrow tips with uppercase first letters should not be redefined.

  If the name contains a hyphen, the second syntax is assumed and everything before the hyphen will be the name used in start arrow specifications, while the text after the hyphen is the name used in end specifications.

- `parameters={`⟨*list of macros*⟩`}`

  As explained earlier, an arrow tip typically needs to be redrawn each time an option like `length` or `inset` is changed. However, for some arrow tips, the `inset` has no influence, while for other it is important whether the arrow is reversed or not. (How keys like `length` actually set TEX dimensions like `\pgfarrowlength` is explained in Section 105.5.)

  The job of the `parameters` key is to specify which dependencies the arrow tip has. Everything that will influence any of the parameters computed in the setup code or used in the drawing code should be listed here.

  The ⟨*list of macros*⟩ will be used inside a `\csname-\endcsname` pair and should expand to the current values of the relevant parameters have. For example, if the arrow tip depends on the current value of `\pgfarrowlength` and `\pgfarrowwidth` only, then ⟨*list of macros*⟩ should be set to `\the\pgfarrowlength,\the\pgfarrowwidth`. (Actually, the comma is optional, the ⟨*list of macros*⟩ does not really have to be a list, just something that can be expanded unambiguously.)

  Note that the line width (`\pgflinewidth`) and the inner line width (`\pgfinnerlinewidth`) are always parameters and need not be specified in the `parameters`.

  It is important to get this parameter right. Otherwise, arrow tips may look wrong because PGF thinks that it can reuse some code when, in reality, this code actually depends on a parameter not listed here.

- `setup code={`⟨*code*⟩`}`

  When an arrow tip is used, the value stored in `parameters` is expanded and it is tested whether the result was encountered before. If not, the ⟨*code*⟩ gets executed (only this once). The code can now do arbitrarily complicated computations the prepare the later drawing of the arrow tip. Also the ⟨*code*⟩ must specify the different tip and back ends and the convex hull points. This is done by calling the following macros inside the ⟨*code*⟩:

  `\pgfarrowssettipend{`⟨*dimension*⟩`}`

  When this command is called inside the setup code of an arrow tip, it specifies that the tip of the drawn arrow will end exactly at ⟨*dimension*⟩. For example, for our earlier example of the large arrow tip, where the tip end was at 1cm, we would call

  ```
  \pgfarrowssettipend{1cm}
  ```

  Note that for efficiency reasons, the ⟨*dimension*⟩ is not passed through `\pgfmathsetlength`; rather what happens is that `\pgf@x=`⟨*dimension*⟩ gets executed. In particular, you can pack further computations into the ⟨*dimension*⟩ by simply starting it with a number and then appending some code that modifies `\pgf@x`. Here is an example where instead of 1cm we use $1\text{cm} - \frac{1}{2}$linewidth as the tip end:

  ```
  \pgfarrowssettipend{1cm\advance\pgf@x by-.5\pgflinewidth}
  ```

  If the command is not called at all inside the setup code, the tip end is set to `0pt`.

  `\pgfarrowssetbackend{`⟨*dimension*⟩`}`

  Works like the command for the tip end, only it sets the back end. In our example we would call

  ```
  \pgfarrowssettipend{-3cm}
  ```

  Defaults to `0pt`.

  `\pgfarrowssetlineend{`⟨*dimension*⟩`}`

  Sets the line end, so in the example we have `\pgfarrowssettipend{-1cm}`. Default to `0pt`.

  `\pgfarrowssetvisualbackend{`⟨*dimension*⟩`}`

  Sets the visual back end, `\pgfarrowssetvisualbackend{-2cm}` in our example. Default to the value of the normal back end.

  `\pgfarrowssetvisualtipend{`⟨*dimension*⟩`}`

  Sets the visual tip end. Default to the value of the normal tip end and, thus, we need not set it in our example.

\pgfarrowshullpoint{⟨*x dimension*⟩}{⟨*y dimension*⟩}

Adds a point to the convex hull of the arrow tip. As for the previous commands, no math parsing is done; instead PGF says \pgf@x=⟨*x dimension*⟩ and then \pgf@y=⟨*y dimension*⟩. Thus, both "dimensions" can contain code for advancing and thus modifying \pgf@x and \pgf@y.

In our example we would write

```
\pgfarrowshullpoint{1cm}{0pt}
\pgfarrowshullpoint{-3cm}{2cm}
\pgfarrowshullpoint{-3cm}{-2cm}
```

\pgfarrowsupperhullpoint{⟨*x dimension*⟩}{⟨*y dimension*⟩}

This command works like the previous command, only it normally adds *two* points to the convex hull: First, the point (⟨*x dimension*⟩, ⟨*y dimension*⟩) and, secondly, the point (⟨*x dimension*⟩, −⟨*y dimension*⟩). However, the second point is only added if the arrow is not a harpoon.

Thus, in our example we could simplify the convex hull to

```
\pgfarrowshullpoint{1cm}{0pt}
\pgfarrowsupperhullpoint{-3cm}{2cm}
```

If the ⟨*y dimension*⟩ is zero or less, only one point, namely (⟨*x dimension*⟩, ⟨*y dimension*⟩), is added to the hull. Thus, we could also have used the upper convex hull command in the first of the two of the above commands.

\pgfarrowssave{⟨*macro*⟩}

As explained earlier, the setup code needs to "communicate" with the drawing code via "saved values". This command get the name of a macro and will store the value this macro had internally. Then, each time drawing code is executed, the value of this macro will be restored.

\pgfarrowssavethe{⟨*register*⟩}

Works like \pgfarrowssave, only the parameter must be a register and \the⟨*register*⟩ will be saved. Typically, you will write something like

```
\pgfarrowssavethe{\pgfarrowlength}
\pgfarrowssavethe{\pgfarrowwidth}
```

To ensure that inside the drawing code the the dimension registers \pgfarrowlength and \pgfarrowwidth are setup with the values they had during the setup.

- drawing code={⟨*code*⟩}

This code will be executed at least once for each setting of the parameters when the time arrow tip is actually drawn. Usually, this one execution will be all and the low-level commands generated inside the ⟨*code*⟩ will we stored in a special cache; but in some cases the ⟨*code*⟩ gets executed each time the arrow tip is used, so do not assume anything about it. Inside the ⟨*code*⟩, you have access to all values that were saved in the setup code as well as to the line width.

The ⟨*code*⟩ should draw the arrow tip "going right along the x-axis". PGF will take care of setting up a canvas transformation beforehand to a rotation such that when the drawing is rendered, the arrow tip that is actually drawn points in the direction of the line. Alternatively, when bending is switched on, even more complicated low-level transformations will be done automatically.

The are some special considerations concerning the ⟨*code*⟩:

- In the ⟨*code*⟩ you may *not* use \pgfusepath since this would try to add arrow tips to the arrow tip and lead to a recursion. Use the "quick" versions \pgfusepathqstroke and so on instead, which never try to add arrow tips.
- If you stroke the path that you construct, you should first set the dashing to solid and set up fixed joins and caps, as needed. This will ensure that the arrow tip will always look the same.
- When the arrow tip code is executed, it is automatically put inside a low-level scope, so nothing will "leak out" from the scope.
- The high-level coordinate transformation matrix will be set to the identity matrix when the code is executed for the first time.

- cache=⟨*true or false*⟩

  When set to `true`, which is the default, the ⟨*code*⟩ will be executed only once for a particular value of parameters and the low-level commands created by the drawing code (using the system layer protocol subsystem, see Section 122) will be cached and reused later on. However, when the drawing code contains "uncacheable" code like a call to `\pgftext`, caching must be switched off by saying `cache=false`.

- bending mode=⟨*mode*⟩

  This key is important only when the `bend` option is used with an arrow, see Section 16.3.8 for an introduction to this option. The `bend` option asks us to, well, bend the arrow head. For some arrow head this is not possible or leads to very strange drawings (for instance, when the `\pgftext` command is used) and then it is better to switch bending off for the arrow head (`flex` will then be used instead). To achieve this, set ⟨*mode*⟩ to `none`.

  For most arrow tips it does, however, make sense to bend them. There are (at least) two different mathematical ways of doing so, see Section 108.4.7 for details. Which of these ways is use can be configured by setting ⟨*mode*⟩ to either `orthogonal` or to `polar`. It is best to try simply try out both when designing an arrow tip to see which works better. Since `orthogonal` is quicker and often gives good or even better results, it is the default. Some arrow tips, however, profit from saying `bending mode=polar`.

- defaults=⟨*arrow keys*⟩

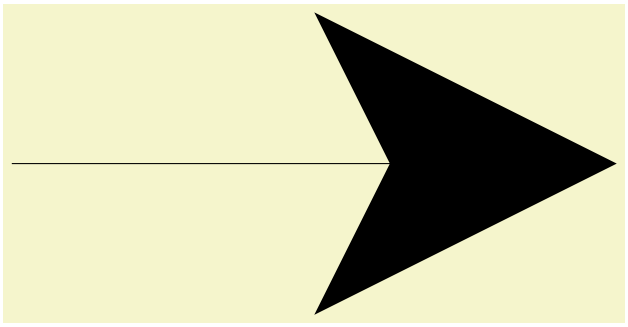
  The ⟨*arrow keys*⟩ allow you to configure the default values for the parameters on which an arrow tip depends. The ⟨*arrow keys*⟩ will be executed first before any other arrow tip options are executed, see Section 16.4.5 for the exact sequence. Also see Section 105.5 below for more details on arrow options.

This concludes the description of the keys you provide for the declaration of an arrow. Let us now have a look at a simple example that uses these features: We want to define an arrow tip kind `foo` that produces the arrow tip we used as our running example. However, to make things a bit more interesting, let us make it "configurable" insofar as the length of the arrow tip can be configured using the `length` option, which sets the `\pgfarrowlength`. By default, this length should be the gigantic 4cm we say in the example, but uses should be able to set it to anything they like. We will not worry about the arrow width or insets, of arrow line width, or harpoons, or anything else in this example to keep it simple.

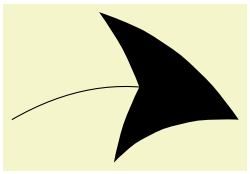
Here is the code:

```
\pgfdeclarearrow{
  name = foo,
  parameters = { \the\pgfarrowlength },
  setup code = {
    % The different end values:
    \pgfarrowssettipend{.25\pgfarrowlength}
    \pgfarrowssetlineend{-.25\pgfarrowlength}
    \pgfarrowssetvisualbackend{-.5\pgfarrowlength}
    \pgfarrowssetbackend{-.75\pgfarrowlength}
    % The hull
    \pgfarrowshullpoint{.25\pgfarrowlength}{0pt}
    \pgfarrowshullpoint{-.75\pgfarrowlength}{.5\pgfarrowlength}
    \pgfarrowshullpoint{-.75\pgfarrowlength}{-.5\pgfarrowlength}
    % Saves: Only the length:
    \pgfarrowssavethe\pgfarrowlength
  },
  drawing code = {
    \pgfpathmoveto{\pgfqpoint{.25\pgfarrowlength}{0pt}}
    \pgfpathlineto{\pgfqpoint{-.75\pgfarrowlength}{.5\pgfarrowlength}}
    \pgfpathlineto{\pgfqpoint{-.5\pgfarrowlength}{0pt}}
    \pgfpathlineto{\pgfqpoint{-.75\pgfarrowlength}{-.5\pgfarrowlength}}
    \pgfpathclose
    \pgfusepathqfill
  },
  defaults = { length = 4cm }
}
```

We can now use it:



```
\usetikzlibrary {arrows.meta}
\tikz \draw [-foo] (0,0) -- (8,0);
```



```
\usetikzlibrary {arrows.meta,bending}
\tikz \draw [-{foo[length=2cm,bend]}] (0,0) to [bend left] (3,0);
```

Defining a Shorthand. The `\pgfdeclarearrow` command can also used to define *shorthands*. This works as follows:

- First, you must provide a `name` just in the same way as when you define a full-flung new arrow tip kind.
- Second, instead of all of the other options listed above, you just use one more option:

  `means`=⟨*end arrow specification*⟩

  This sets up things so that whenever ⟨*name*⟩ is now used in an arrow specification, it will be replaced by the ⟨*end arrow specification*⟩ (the problems resulting form the ⟨*name*⟩ begin used in a start arrow specification are taken care of automatically). See also Section 16.4.4 for details on the order in which options get executed in such cases.

  Note that the ⟨*end arrow specification*⟩ will be executed immediately to build the so-called arrow option caches, a concept explored in more detail in Section 105.5.4. In practice, this has mainly two effects: First, all arrow tips referred to in the specification must already exist (at least as "dummy" versions). Second, all dimensions mentioned in options of the ⟨*end arrow specification*⟩ will be evaluated immediately. For instance, when you write

  ```
  \pgfdeclarearrow{ name=foo, means = bar[length=2cm+\mydimen] }
  ```

  The value `2cm+\mydimen` is evaluated immediately. When `foo` is used later on and `\mydimen` has changed, this has no effect.

105.5 Handling Arrow Options

When you declare an arrow tip, your drawing code should take into account the different arrow keys set for it (like the arrow tip length, width, or harpooning). The different arrow keys that are available have been described in detail in Section 16.3; but how do we access the values set by an option like `length` or `harpoon` or `bend` in the drawing code? In the present section we have a look at how this works.

105.5.1 Dimension Options

Most arrow keys, like `length` or `width'`, simple set a TEX dimension register to a certain value. For example, `length` sets the value of the TEX dimension register `\pgfarrowlength`. Note that `length` takes several values as input with a complicated semantics as explained for the `length` key on page 192. All of these settings are not important for the setup code: When it gets executed, the code behind the `length` key will have computed a simple number that is stored in `\pgfarrowlength`. Indeed, inside the setup code you do not have access to the exact value given to the `length` key; just to the final computed value.

The following TEX dimensions are available to the setup code:

- `\pgfarrowslength`. It gets set by the arrow keys `length` and `angle`.
- `\pgfarrowswidth`. It gets set by `width`, `width'`, and `angle`.
- `\pgfarrowsinset`. It gets set by `inset` and `inset'`.
- `\pgfarrowslinewidth`. It gets set by `line width` and `line width'`.

If your setup code depends on any of them, add them to the `parameters` key of the arrow tip.

105.5.2 True–False Options

A number of arrow keys just do a yes/no switch, like `reversed`. All of them setup a TEX-if that you can access in the setup code:

- `\ifpgfarrowreversed` is setup by `reversed`.
- `\ifpgfarrowswap` is setup by `swap` and also `right`.
- `\ifpgfarrowharpoon` is setup by `harpoon` and also `left` and `right`.
- `\ifpgfarrowroundcap` is set to true by `line cap=round` and set to false by `line cap=butt`. It also gets (re)set by `round` and `sharp`.
- `\ifpgfarrowroundjoin` is set to true by `line join=round` and set to false by `line join=miter`. It also gets (re)set by `round` and `sharp`.
- `\ifpgfarrowopen` is set to true by `fill=none` and by `open` (which is a shorthand for `fill=none`) and set to false by `color` and all other `fill=`⟨*color*⟩.

If you code depends on any of these, you must add them to the `parameters` in such a way that the parameters are different when the TEX-if is set from when it is not set. An easy way to achieve this is to write something like

```
parameters = { \the\pgfarrowlength,...,
               \ifpgfarrowharpoon h\fi\
               \ifpgfarrowroundjoin j\fi}
```

In other words, for each set parameter on which the arrow tip depends, a specific letter is added to the parameters, making them unique.

The first two of the above keys are a bit special: Reversing and swapping an arrow tip can be done just by fiddling with the transformation matrix: a reverse is a "flip" along the y-axis and a swap is a flip along the x-axis. This is done automatically by PGF.

Nevertheless, you may wish to modify you code in dependence especially of the `reverse` key: When `\ifpgfarrowreverse` is true, PGF will flip the coordinate system along the y-axis, will negate all end values (like line end, tip end, and so on) and will exchange the meaning of back end and tip end as well as of visual back end and visual back end. Usually, this is exactly what one need; *except* that the line end may no longer be appropriate. After all, the line end should be chosen so that it is completely covered by the arrow. Now, when the arrow tip is open, a reversed arrow should no longer have the line end near the old visual back end, but near to the old visual tip end.

For these reasons, you may need to make the computation of the line end dependent on whether the arrow is reversed or not. Note that when you specify a different line end for a reversed arrow tip, the transformation and inverting of the coordinate system will still be done, meaning that if `reverse` is true, you need to specify a line end in the "old" coordinate system that is at the position where, after everything is inverted, it will be at the correct position. Usually that means that if the `reverse` option is set, you need to *increase* the line end.

105.5.3 Inaccessible Options

There are some options that influence the way an arrow tip looks, but that you cannot access inside the setup code. Handling these options lies entirely with PGF. If you wish your setup code to handle these options, you have to setup your own "parallel" options.

- `quick`, `flex`, `flex'`, and `bend` are all handled automatically. You can, however, set the `bending mode` to avoid bending of your arrow tip.

- The colors set by `color` and `fill`. You can, however, access them indirectly, namely through the current stroke and fill colors.
- `sep`

105.5.4 Defining New Arrow Keys

The set of predefined options is already quite long and most arrow tips will not need more than the predefined options. However, sometimes an arrow tip may need to introduce a new special-purpose option. For instance, suppose we wish to introduce a new fictive arrow key `depth`. In such cases, you must do two things:

1. Introduce a new dimension register or macro that will hold the configuration value and which will be accessed by the setup code. The could be achieved by saying

   ```
   \newdimen\pgfarrowdepth
   ```

2. Introduce a new arrow key option `/pgf/arrow keys/depth` that allows users to configure the new macro or register.

When an arrow is selected via for instance `foo[depth=5pt]`, the key–value pairs between the square brackets are executed with the path prefix `/pgf/arrow keys`. Thus, in the example, our depth key would get executed. Thus, it is tempting to write something like

```
\pgfkeys{/pgf/arrow keys/depth/.code = \pgfmathsetlength{\pgfarrowdepth}{#1}}
```

Sadly, this will not work. The reason is that there is yet another level of caching involved when PGF processes arrow tips: The option cache! The problem is each time an arrow tip is used, even when the drawing code of the arrow tip is nicely cached, we still need to process the options in `foo[length=5pt]` to find out which version in the cache we would like to access. To make matters worse, `foo` might be a shorthand that calls other arrow tips, which add more options, and so on. Unfortunately, executing keys is quite an expensive operation (PGF's key–value parser is powerful, but that power comes at a price). So, whenever possible, we do *not* want the key–value parser to be started.

For these reasons, when something like `foo[`⟨*options*⟩`]` is encountered inside a shorthand, the ⟨*options*⟩ are executed only once. They should now setup the *arrow option cache*, which is some code that, when executed, should setup the values that the ⟨*options*⟩ configure. In our example, the `depth` key should add something to the arrow option cache that sets `\pgfarrowdepth` to the given value.

Adding something to the arrow option cache is done using the following command:

`\pgfarrowsaddtooptions{`⟨*code*⟩`}`

This command should be called by keys with the prefix `/pgf/arrow keys` to add code to the arrow option cache. For our `depth` key example, we could use this key as follows:

```
\pgfkeys{/pgf/arrow keys/depth/.code=
  \pgfarrowsaddtooptions{\pgfmathsetlength{\pgfarrowdepth}{#1}}
```

Actually, this is still not optimal since the expensive `\pgfmathsetlength` command is now called each time an arrow tip is used with the `depth` option set. The trick is to do the expensive operation only once and then store only very quick code in the arrow option cache:

```
\pgfkeys{/pgf/arrow keys/depth/.code=
  \pgfmathsetlength{\somedimen}{#1}
  \pgfarrowsaddtooptions{\pgfarrowdepth=\somedimen} % buggy
```

The above code will not (yet) work since `\somedimen` will surely have a different value when the cache is executed. The trick is to use some `\expandafter`s:

```
\pgfkeys{/pgf/arrow keys/depth/.code=
  \pgfmathsetlength{\somedimen}{#1}
  \expandafter\pgfarrowsaddtooptions\expandafter{\expandafter\pgfarrowdepth\expandafter=\the\somedimen}
```

\pgfarrowsaddtolateoptions{⟨*code*⟩}

This command works like \pgfarrowsaddtooptions, only the ⟨*code*⟩ will be executed "later" than the code added by the normal version of the command. This is useful for keys that depend on the length of an arrow: Keys like width' want to define the arrow width as a multiple of the arrow length, but when the width' key is given, the length may not yet have been specified. By making the computation of the width a "late" option, we ensure that \pgfarrowlength will have been setup correctly.

If you define a new option that sets a dimensions and if that dimension should change in accordance to the setting of either scale length or scale width, you need to make PGF "aware" of this using the following key:

\pgfarrowsaddtolengthscalelist{⟨*dimension register*⟩}

Each time an arrow tip is used, the given ⟨*dimension register*⟩ will be multiplied by the scale length factor prior to the actual drawing. You call this command only once in the preamble somewhere.

\pgfarrowsaddtowidthscalelist{⟨*dimension register*⟩}

Works like \pgfarrowsaddtolengthscalelist, only for width parameters.

\pgfarrowsthreeparameters{⟨*line-width dependent size specification*⟩}

This command is useful for parsing the values given to keys like length or width the expect a dimension followed optionally for some numbers. This command converts the ⟨*line-width dependent size specification*⟩, which may consist of one, two, or three numbers, into a triple of three numbers in curly braces, which gets stored in the macro \pgfarrowstheparameters. Here is an example, where \showvalueofmacro is used in this example to show the value stored in a macro:

```
{2.0pt}{1}{0}
```

```
\pgfarrowsthreeparameters{2pt 1}
\showvalueofmacro\pgfarrowstheparameters
```

\pgfarrowslinewidthdependent{⟨*dimension*⟩}{⟨*line width factor*⟩}{⟨*outer factor*⟩}

This command takes three parameters and does the "line width dependent computation" described on page 192 for the length key. The result is returned in \pgf@x.

The idea is that you can setup line-width dependent keys like length or width using code like the following:

```
\pgfkeys{/pgf/arrow keys/depth/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions\expandafter{%
    \expandafter\pgfarrowslinewidthdependent\pgfarrowstheparameters% compute...
    \pgfarrowdepth\pgf@x% ... and store.
  }%
}
```

\pgfarrowslengthdependent{⟨*dimension*⟩}{⟨*length factor*⟩}{⟨*dummy*⟩}

This command takes three parameters, of which the last one is ignored, and does the "length dependent computation" described for the width' and inset' keys. The result is returned in \pgf@x.

You can setup length dependent keys using code like the following:

```
\pgfkeys{/pgf/arrow keys/depth'/.code={%
  \pgfarrowsthreeparameters{#1}%
  \expandafter\pgfarrowsaddtolateoptions\expandafter{%
    \expandafter\pgfarrowslengthdependent\pgfarrowstheparameters% compute...
    \pgfarrowdepth\pgf@x% ... and store.
  }%
}
```

106 Nodes and Shapes

This section describes the `shapes` module.

```
\usepgfmodule{shapes}  % LaTeX and plain TeX when using pure pgf
\usepgfmodule[shapes]  % ConTeXt when using pure pgf
```

This module defines commands both for creating nodes and for creating shapes. The package is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

106.1 Overview

PGF comes with a sophisticated set of commands for creating *nodes* and *shapes*. A *node* is a graphical object that consists (typically) of (one or more) text labels and some additional stroked or filled paths. Each node has a certain *shape*, which may be something simple like a `rectangle` or a `circle`, but it may also be something complicated like a `uml class diagram` (this shape is currently not implemented, though). Different nodes that have the same shape may look quite different, however, since shapes (need not) specify whether the shape path is stroked or filled.

106.1.1 Creating and Referencing Nodes

You create a node by calling the macro `\pgfnode` or the more general `\pgfmultipartnode`. This macro takes several parameters and draws the requested shape at a certain position. In addition, it will "remember" the node's position within the current `{pgfpicture}`. You can then, later on, refer to the node's position. Coordinate transformations are "fully supported", which means that if you used coordinate transformations to shift or rotate the shape of a node, the node's position will still be correctly determined by PGF. This is *not* the case if you use canvas transformations instead.

106.1.2 Anchors

An important property of a node or a shape in general are its *anchors*. Anchors are "important" positions in a shape. For example, the `center` anchor lies at the center of a shape, the `north` anchor is usually "at the top, in the middle" of a shape, the `text` anchor is the lower left corner of the shape's text label (if present), and so on.

Anchors are important both when you create a node and when you reference it. When you create a node, you specify the node's "position" by asking PGF to place the shape in such a way that a certain anchor lies at a certain point. For example, you might ask that the node is placed such that the `north` anchor is at the origin. This will effectively cause the node to be placed below the origin.

When you reference a node, you always reference an anchor of the node. For example, when you request the "`north` anchor of the node just placed" you will get the origin. However, you can also request the "`south` anchor of this node", which will give you a point somewhere below the origin. When a coordinate transformation was in force at the time of creation of a node, all anchors are also transformed accordingly.

106.1.3 Layers of a Shape

The simplest shape, the `coordinate`, has just one anchor, namely the `center`, and a label (which is usually empty). More complicated shapes like the `rectangle` shape also have a *background path*. This is a PGF-path that is defined by the shape. The shape does not prescribe what should happen with the path: When a node is created, this path may be stroked (resulting in a frame around the label), filled (resulting in a background color for the text), or just discarded.

Although most shapes consist just of a background path plus some label text, when a shape is drawn, up to seven different layers are drawn:

1. The "behind the background layer". Unlike the background path, which can be used in different ways by different nodes, the graphic commands given for this layer will always stroke or always fill the path they construct. They might also insert some text that is "behind everything".

2. The background path layer. How this path is used depends on the arguments of the `\pgfnode` command.

3. The “before the background path layer”. This layer works like the first one, only the commands of this layer are executed after the background path has been used (in whatever way the creator of the node chose).
4. The label layer. This layer inserts the node’s text box(es).
5. The “behind the foreground layer”. This layer, like the first layer, once more contains graphic commands that are “simply executed”.
6. The foreground path layer. This path is treated in the same way as the background path, only it is drawn after the label text has been drawn.
7. The “before the foreground layer”.

Which of these layers are actually used depends on the shape.

106.1.4 Node Parts

A shape typically does not consist only of different background and foreground paths, but it may also have text labels. Indeed, for many shapes the text labels are the more important part of the shape.

Most shapes will have only one text label. In this case, this text label is simply passed as a parameter to the `\pgfnode` command. When the node is drawn, the text label is shifted around such that its lower left corner is at the `text` anchor of the node.

More complicated shapes may have more than one text label. Nodes of such shapes are called *multipart nodes*. The different *node parts* are simply the different text labels. For example, a `uml class` shape might have a `class name` part, a `method` part and an `attributes` part. Indeed, single part nodes are a special case of multipart nodes: They only have one part named `text`.

When a shape is declared, you must specify the node parts. There is a simple command called `\nodeparts` that takes a list of the part names as input. When you create a node of a multipart shape, for each part of the node you must have set up a TEX-box containing the text of the part. For a part named `XYZ` you must set up the box `\pgfnodepartXYZbox`. The box will be placed at the anchor `XYZ`. See the description of `\pgfmultipartnode` for more details.

106.2 Creating Nodes

106.2.1 Creating Simple Nodes

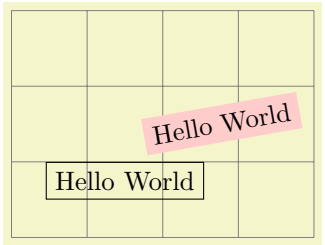
`\pgfnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*label text*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

This command creates a new node. The ⟨*shape*⟩ of the node must have been declared previously using `\pgfdeclareshape`.

The shape is shifted such that the ⟨*anchor*⟩ is at the origin. In order to place the shape somewhere else, use the coordinate transformation prior to calling this command.

The ⟨*name*⟩ is a name for later reference. If no name is given, nothing will be “saved” for the node, it will just be drawn.

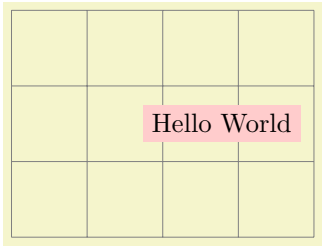
The ⟨*path usage command*⟩ is executed for the background and the foreground path (if the shape defines them).




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \pgftransformshift{\pgfpoint{1.5cm}{1cm}}
    \pgfnode{rectangle}{north}{Hello World}{hellonode}{\pgfusepath{stroke}}
  }
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

As can be seen, all coordinate transformations are also applied to the text of the shape. Sometimes, it is desirable that the transformations are applied to the point where the shape will be

anchored, but you do not wish the shape itself to be transformed. In this case, you should call `\pgftransformresetnontranslations` prior to calling the `\pgfnode` command.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (4,3);
  {
    \color{red!20}
    \pgftransformrotate{10}
    \pgftransformshift{\pgfpoint{3cm}{1cm}}
    \pgftransformresetnontranslations
    \pgfnode{rectangle}{center}
      {\color{black}Hello World}{hellonode}{\pgfusepath{fill}}
  }
\end{tikzpicture}
```

The ⟨*label text*⟩ is typeset inside the TEX-box `\pgfnodeparttextbox`. This box is shown at the `text` anchor of the node, if the node has a `text` part. See the description of `\pgfmultipartnode` for details.

106.2.2 Creating Multi-Part Nodes

`\pgfmultipartnode{`⟨*shape*⟩`}{`⟨*anchor*⟩`}{`⟨*name*⟩`}{`⟨*path usage command*⟩`}`

This command is the more general (and less user-friendly) version of the `\pgfnode` command. While the `\pgfnode` command can only be used for shapes that have a single part (which is the case for most shapes), this command can also be used with multi-part nodes.

When this command is called, for each node part of the node you must have set up one TEX-box. Suppose the shape has two parts: The `text` part and the `lower` part. Then, prior to calling `\pgfmultipartnode`, you must have set up the boxes `\pgfnodeparttextbox` and `\pgfnodepartlowerbox`. These boxes may contain any TEX-text. The shape code will then compute the positions of the shape's anchors based on the sizes of the these shapes. Finally, when the node is drawn, the boxes are placed at the anchor positions `text` and `lower`.




```
\usetikzlibrary {shapes}
\setbox\pgfnodeparttextbox=\hbox{$q_1$}
\setbox\pgfnodepartlowerbox=\hbox{01}
\begin{pgfpicture}
  \pgfmultipartnode{circle split}{center}{my state}{\pgfusepath{stroke}}
\end{pgfpicture}
```

Note: Be careful when using the `\setbox` command inside a `{pgfpicture}` command. You will have to use `\pgfinterruptpath` at the beginning of the box and `\endpgfinterruptpath` at the end of the box to make sure that the box is typeset correctly. In the above example this problem was sidestepped by moving the box construction outside the environment.

Note: It is not necessary to use `\newbox` for every node part name. Although you need a different box for each part of a single shape, two different shapes may very well use the same box even when the names of the parts are different. Suppose you have a `circle split` shape that has a `lower` part and you have a `uml class` shape that has a `methods` part. Then, in order to avoid exhausting TEX's limited number of box registers, you can say

```
\newbox\pgfnodepartlowerbox
\let\pgfnodepartmethodsbox=\pgfnodepartlowerbox
```

Also, when you have a node part name with spaces like `class name`, it may be useful to create an alias:

```
\newbox\mybox
\expandafter\let\csname pgfnodepartclass namebox\endcsname=\mybox
```

`\pgfcoordinate{`⟨*name*⟩`}{`⟨*coordinate*⟩`}`

This command creates a node of shape `coordinate` at the given ⟨*coordinate*⟩. Exactly the same effect can be achieved using first a shift of the coordinate system to ⟨*coordinate*⟩, followed by creating a node of shape `coordinate` named ⟨*name*⟩. However, this command is easier and more natural to use and, more importantly, it is much faster.

\pgfnodealias{⟨*new name*⟩}{⟨*existing node*⟩}

This command does not actually create a new node. Rather, it allows you to subsequently access the node ⟨*existing node*⟩ using the name ⟨*new name*⟩.

\pgfnoderename{⟨*new name*⟩}{⟨*existing node*⟩}

This command renames an existing node.

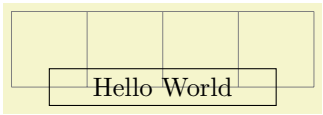
There are a number of values that have an influence on the size of a node. These values are stored in the following keys.

/pgf/minimum width=⟨*dimension*⟩ (no default, initially 1pt)

alias /tikz/minimum width

This key stores the *recommended* minimum width of a shape. Thus, when a shape is drawn and when the shape's width would be smaller than ⟨*dimension*⟩, the shape's width is enlarged by adding some empty space.

Note that this value is just a recommendation. A shape may choose to ignore this key.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfset{minimum width=3cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

/pgf/minimum height=⟨*dimension*⟩ (no default, initially 1pt)

alias /tikz/minimum height

Works like /pgf/minimum width.

/pgf/minimum size=⟨*dimension*⟩ (no default)

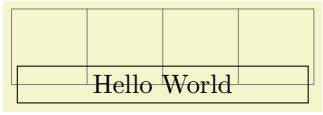
alias /tikz/minimum size

This style both /pgf/minimum width and /pgf/minimum height to ⟨*dimension*⟩.

/pgf/inner xsep=⟨*dimension*⟩ (no default, initially 0.3333em)

alias /tikz/inner xsep

This key stores the *recommended* horizontal inner separation between the label text and the background path. As before, this value is just a recommendation and a shape may choose to ignore this key.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfset{inner xsep=1cm}
  \pgfnode{rectangle}{center}{Hello World}{}{\pgfusepath{stroke}}
\end{tikzpicture}
```

/pgf/inner ysep=⟨*dimension*⟩ (no default, initially 0.3333em)

alias /tikz/inner ysep

Works like /pgf/inner xsep.

/pgf/inner sep=⟨*dimension*⟩ (no default)

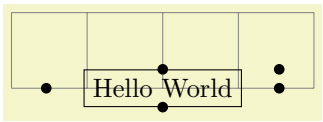
alias /tikz/inner sep

This style sets both /pgf/inner xsep and /pgf/inner ysep to ⟨*dimension*⟩.

/pgf/outer xsep=⟨*dimension*⟩ (no default, initially .5\pgflinewidth)

alias /tikz/outer xsep

This key stores the recommended horizontal separation between the background path and the "outer anchors". For example, if ⟨*dimension*⟩ is 1cm then the east anchor will be 1cm to the right of the right border of the background path. As before, this value is just a recommendation.




```
\begin{tikzpicture}
  \draw[help lines] (-2,0) grid (2,1);

  \pgfset{outer xsep=.5cm}
  \pgfnode{rectangle}{center}{Hello World}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{tikzpicture}
```

/pgf/outer ysep=⟨*dimension*⟩ (no default, initially `.5\pgflinewidth`)
alias `/tikz/outer ysep`

Works like `/pgf/outer xsep`.

/pgf/outer sep=⟨*dimension*⟩ (no default)
alias `/tikz/outer sep`

This style sets both `/pgf/outer xsep` and `/pgf/outer ysep` to ⟨*dimension*⟩.

106.2.3 Deferred Node Positioning

Normally, when a node is created using a command like `\pgfnode`, the node is immediately inserted into the current picture. In particular, you have no chance to change the position of a created node after it has been created. Using `\pgfpositionnodelater` in concert with `\pgfpositionnodenow`, you can create a node whose position is determined only at some later time.

`\pgfpositionnodelater{`⟨*macro name*⟩`}`

This command is not a replacement for `\pgfnode`. Rather, when this command is used in a scope, all subsequent node creations in this scope will be affected in the following way: When a node is created, it is not inserted into the current picture. Instead, it is stored in the box `\pgfpositionnodelaterbox`. Furthermore, the node is not relevant for the picture's bounding box, but a bounding box for the node is computed and stored in the macros `\pgfpositionnodelaterminx` to `\pgfpositionnodelatermaxy`. Then, the ⟨*macro name*⟩ is called with the following macros set up:

`\pgfpositionnodelaterbox`

A box register number (`0` currently) that stores the node's paths and texts. You should move the contents of this box to a box of your choice inside ⟨*macro name*⟩.

`\pgfpositionnodelatername`

The name of the just-created-node. This name will be the originally "desired" name of the box plus the fixed prefix `not yet positionedPGFINTERNAL`. The idea is to ensure that the original name is not inadvertently used before the node is actually positioned. When `\pgfpositionnodenow` is called, it will change the name to the original name.

`\pgfpositionnodelaterminx`

The minimal x-position of a bounding box of the node. This bounding box refers to the node when it is positioned with the anchor at the origin. It is guaranteed, that this macro will contain a dimension in the format ⟨*number*⟩`pt`.

`\pgfpositionnodelaterminy`

`\pgfpositionnodelatermaxx`

`\pgfpositionnodelatermaxy`

Once a late node has been created, you can add arbitrary code in the same picture. Then, at some later point, you call `\pgfpositionnodenow` to finally position the node at a given position. At this point, the above macros must have the exact same values they had when ⟨*macro name*⟩ was called. Note that

the above macros are local to a scope that ends right after the call to ⟨*macro name*⟩, so it is your job to copy the values to safety inside ⟨*macro name*⟩.

The following two macros will also be set inside the call to ⟨*macro name*⟩, but they are only "informative" in the sense that you need *not* restore these macros when `\pgfpositionnodenow` is called.

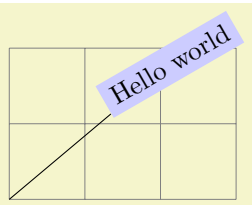
`\pgfpositionnodelaterpath`

This macro stores the path of the background of the node. See Section 121 for an overview of how these paths are encode.

By setting ⟨*macro name*⟩ to `\relax` (which is the default), you can switch off the whole mechanism. When a picture is interrupted, this is done automatically.

`\pgfpositionnodenow{`⟨*coordinate*⟩`}`

This command is used to position a node that has previously been created using the command `\pgfpositionnodelater`. When `\pgfpositionnodenow` is called, the macros and boxes mentioned in the description of `\pgfpositionnodenow` must be set to the value they had when the ⟨*macro name*⟩ was called. Provided this is the case, this command will insert the box into the current picture, shifted by ⟨*coordinate*⟩. Then, the late code (see below) is called. Subsequently, you can refer to the node with its original name as if it had just been created.




```
\newbox\mybox

\def\mysaver{
  \global\setbox\mybox=\box\pgfpositionnodelaterbox
  \global\let\myname=\pgfpositionnodelatername
  \global\let\myminx=\pgfpositionnodelaterminx
  \global\let\myminy=\pgfpositionnodelaterminy
  \global\let\mymaxx=\pgfpositionnodelatermaxx
  \global\let\mymaxy=\pgfpositionnodelatermaxy
}

\begin{tikzpicture}
  {
    \pgfpositionnodelater{\mysaver}
    \node [fill=blue!20,below,rotate=30] (hi) {Hello world};
  }
  \draw [help lines] (0,0) grid (3,2);

  \let\pgfpositionnodelatername=\myname
  \let\pgfpositionnodelaterminx=\myminx
  \let\pgfpositionnodelaterminy=\myminy
  \let\pgfpositionnodelatermaxx=\mymaxx
  \let\pgfpositionnodelatermaxy=\mymaxy
  \setbox\pgfpositionnodelaterbox=\box\mybox
  \pgfpositionnodenow{\pgfqpoint{2cm}{2cm}}

  \draw (hi) -- (0,0);
\end{tikzpicture}
```

`\pgfnodepostsetupcode{`⟨*node name*⟩`}{`⟨*code*⟩`}`

When you call this macro inside a scope for which the `\pgfpositionnodelater` has been called, the ⟨*code*⟩ will be stored internally. Later, when the node named ⟨*node name*⟩ is actually positioned using `\pgfpositionnodenow`, the ⟨*code*⟩ will be executed. When this macro is called multiple times with the same ⟨*node name*⟩, the ⟨*code*⟩ accumulates. However, When `\pgfpositionnodenow` is called, the code stored for the node is cleared.

The main purpose of this mechanism is to allow Ti*k*Z to store so-called "late options" with a node that will be positioned only later.

106.3 Using Anchors

Each shape defines a set of anchors. We saw already that the anchors are used when the shape is drawn: the shape is placed in such a way that the given anchor is at the origin (which in turn is typically translated somewhere else).

One has to look up the set of anchors of each shape, there is no "default" set of anchors, except for the `center` anchor, which should always be present. Also, most shapes will declare anchors like `north` or `east`, but this is not guaranteed.

106.3.1 Referencing Anchors of Nodes in the Same Picture

Once a node has been defined, you can refer to its anchors using the following commands:

`\pgfpointanchor{`⟨*node*⟩`}{`⟨*anchor*⟩`}`

This command is another "point command" like the commands described in Section 101. It returns the coordinate of the given ⟨*anchor*⟩ in the given ⟨*node*⟩. The command can be used in commands like `\pgfpathmoveto`.




```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

  \pgfpathcircle{\pgfpointanchor{x}{north}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{south}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{east}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{west}}{2pt}
  \pgfpathcircle{\pgfpointanchor{x}{north east}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

In the above example, you may have noticed something curious: The rotation transformation is still in force when the anchors are invoked, but it does not seem to have an effect. You might expect that the rotation should apply to the already rotated points once more.

However, `\pgfpointanchor` returns a point that takes the current transformation matrix into account: *The inverse transformation to the current coordinate transformation is applied to an anchor point before returning it.*

This behavior may seem a bit strange, but you will find it very natural in most cases. If you really want to apply a transformation to an anchor point (for example, to "shift it away" a little bit), you have to invoke `\pgfpointanchor` without any transformations in force. Here is an example:




```
\begin{pgfpicture}
  \pgftransformrotate{30}
  \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}

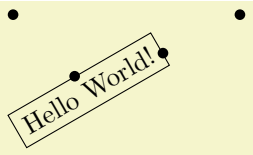
  {
    \pgftransformreset
    \pgfpointanchor{x}{east}
    \xdef\mycoordinate{\noexpand\pgfpoint{\the\pgf@x}{\the\pgf@y}}
  }

  \pgfpathcircle{\mycoordinate}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

A special situation arises when the ⟨*node*⟩ lies in a picture different from the current picture. In this case, if you have not told PGF that the picture should be "remembered", the ⟨*node*⟩ will be treated as if it lay in the current picture. For example, if the ⟨*node*⟩ was at position $(3, 2)$ in the original picture, it is treated as if it lay at position $(3, 2)$ in the current picture. However, if you have told PGF to remember the picture position of the node's picture and also of the current picture, then `\pgfpointanchor` will return a coordinate that corresponds to the position of the node's anchor on the page, transformed into the current coordinate system. For examples and more details see Section 106.3.2.

`\pgfpointshapeborder{`⟨*node*⟩`}{`⟨*point*⟩`}`

This command returns the point on the border of the shape that lies on a straight line from the center of the node to ⟨*point*⟩. For complex shapes it is not guaranteed that this point will actually lie on the border, it may be on the border of a "simplified" version of the shape.




```
\begin{pgfpicture}
  \begin{pgfscope}
    \pgftransformrotate{30}
    \pgfnode{rectangle}{center}{Hello World!}{x}{\pgfusepath{stroke}}
  \end{pgfscope}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{2cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{2cm}{1cm}}{2pt}
  \pgfpathcircle{\pgfpointshapeborder{x}{\pgfpoint{-1cm}{1cm}}}{2pt}
  \pgfpathcircle{\pgfpoint{-1cm}{1cm}}{2pt}
  \pgfusepath{fill}
\end{pgfpicture}
```

Remark: If the given ⟨*point*⟩ is almost identical to the center of ⟨*node*⟩, the node center is returned and a warning message will be printed.

106.3.2 Referencing Anchors of Nodes in Different Pictures

As a picture is typeset, PGF keeps track of the positions of all nodes inside the picture. What PGF does not remember is the position of the picture *itself* on the page. Thus, if you define a node in one picture and then try to reference this node while another picture is typeset, PGF will only know the position of the nodes that you try to typeset inside the original picture, but it will not know where this picture lies. What is missing is the relative positioning of the two pictures.

To overcome this problem, you need to tell PGF that it should remember the position of pictures on a page. If these positions are remembered, then PGF can compute the offset between the pictures and make nodes in different pictures accessible.

Determining the positions of pictures on the page is, alas, not-so-easy. Because of this, PGF does not do so automatically. Rather, you have to proceed as follows:

1. You have to use a backend driver that supports position tracking. pdfTEX is one such driver, `dvips` currently is not.
2. You have to say `\pgfrememberpicturepositiononpagetrue` somewhere before or inside every picture
   - in which you wish to reference a node and
   - from which you wish to reference a node in another picture.

   The second item is important since PGF does not only need to know the position of the picture in which the node you wish to reference lies, but it also needs to know where the current picture lies.
3. You typically have to run TEX twice (depending on the backend driver) since the position information typically gets written into an external file on the first run and is available only on the second run.
4. You have to switch off automatic bounding bound computations. The reason is that the node in the other picture should not influence the size of the bounding box of the current picture. You should say `\pgfusepath{use as bounding box}` before using a coordinate in another picture.

106.4 Special Nodes

There are several special nodes that are always defined and which you should not attempt to redefine.

Predefined node `current bounding box`

This node is of shape `rectangle`. Unlike normal nodes, its size changes constantly and always reflects the size of the bounding box of the current picture. This means that, for instance, that

```
\pgfpointanchor{current bounding box}{south east}
```

returns the lower left corner of the bounding box of the current picture.

Predefined node `current path bounding box`

This node is also of shape `rectangle`. Its size is the size of the bounding box of the current path.

Predefined node `current subpath start`

This node is of shape `coordinate` and is at the beginning of the current subpath. This is the position of the last move-to operation.

Predefined node `current page`

This node is inside a virtual remembered picture. The size of this node is the size of the current page. This means that if you create a remembered picture and inside this picture you reference an anchor of this node, you reference an absolute position on the page. To demonstrate the effect, the following code puts some text in the lower left corner of the current page. Note that this works only if the backend driver supports it, otherwise the text is inserted right here.

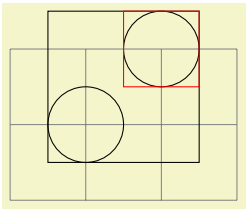
```
\pgfrememberpicturepositiononpagetrue
\begin{pgfpicture}
  \pgfusepath{use as bounding box}
  \pgftransformshift{\pgfpointanchor{current page}{south west}}
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \pgftext[left,base]{
    \textcolor{red}{
      Text absolutely positioned in
      the lower left corner.}
  }
\end{pgfpicture}
```

There is also an option that allows you to create new special nodes quite similar to the above:

`/pgf/local bounding box`=⟨*node name*⟩ (no default)

alias `/tikz/local bounding box`

This defines a new node ⟨*node name*⟩ whose size is the bounding box around all objects in the current scope starting at the position where this option was given. After the end of the scope, the ⟨*node name*⟩ is still available. You can use this option to keep track of the size of a certain area. Note that excessive use of this option (keeping track of dozens of bounding boxes at the same time) will slow things down.



```
\usetikzlibrary {scopes}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  { [local bounding box=outer box]
    \draw (1,1) circle [radius=.5]
      [local bounding box=inner box] (2,2) circle [radius=.5];
  }
  \draw      (outer box.south west) rectangle (outer box.north east);
  \draw[red] (inner box.south west) rectangle (inner box.north east);
\end{tikzpicture}
```

106.5 Declaring New Shapes

There are only three predefined shapes, see Section 71.2, so there must be some way of defining new shapes. Defining a shape is, unfortunately, a not-quite-trivial process. The reason is that shapes need to be both very flexible (their size will vary greatly according to circumstances) and they need to be constructed reasonably "fast". PGF must be able to handle pictures with several hundreds of nodes and documents with thousands of nodes in total. It would not do if PGF had to compute and store, say, dozens of anchor positions for every node.

106.5.1 What Must Be Defined For a Shape?

In order to define a new shape, you must provide:

- a *shape name*,
- code for computing the *saved anchors* and *saved dimensions*,
- code for computing *anchor* positions in terms of the saved anchors,
- optionally code for the *background path* and *foreground path*,
- optionally code for *things to be drawn before or behind* the background and foreground paths.
- optionally a list of node parts.

Text absolutely positioned in the lower left corner.

106.5.2 Normal Anchors Versus Saved Anchors

Anchors are special places in a shape. For example, the `north east` anchor, which is a normal anchor, lies at the upper right corner of the `rectangle` shape, as does `\northeast`, which is a saved anchor. The difference is the following: *saved anchors are computed and stored for each node, anchors are only computed as needed.* The user only has access to the normal anchors, but a normal anchor can just "copy" or "pass through" the location of a saved anchor.

The idea behind all this is that a shape can declare a very large number of normal anchors, but when a node of this shape is created, these anchors are not actually computed. However, this causes a problem: When we wish to reference an anchor of a node at some later time, we must still be able to compute the position of the anchor. For this, we may need a lot of information: What was the transformation matrix that was in force when the node was created? What was the size of the text box? What were the values of the different separation dimensions? And so on.

To solve this problem, PGF will always compute the locations of all *saved anchors* and store these positions. Then, when an normal anchor position is requested later on, the anchor position can be given just from knowing where the locations of the saved anchors are.

As an example, consider the `rectangle` shape. For this shape two anchors are saved: The `\northeast` corner and the `\southwest` corner. A normal anchor like `north west` can now easily be expressed in terms of these coordinates: Take the x-position of the `\southwest` point and the y-position of the `\northeast` point. The `rectangle` shape currently defines 13 normal anchors, but needs only two saved anchors. Adding new anchors like a `south south east` anchor would not increase the memory and computation requirements of pictures.

All anchors (both saved and normal) are specified in a local *shape coordinate space*. This is also true for the background and foreground paths. The `\pgfnode` macro will automatically apply appropriate transformations to the coordinates so that the shape is shifted to the right anchor or otherwise transformed.

106.5.3 Command for Declaring New Shapes

The following command declares a new shape:

`\pgfdeclareshape{`⟨*shape name*⟩`}{`⟨*shape specification*⟩`}`

This command declares a new shape named ⟨*shape name*⟩. The shape name can later be used in commands like `\pgfnode`.

The ⟨*shape specification*⟩ is some TEX code containing calls to special commands that are only defined inside the ⟨*shape specification*⟩ (similarly to commands like `\draw` that are only available inside the `{tikzpicture}` environment).

Example: Here is the code of the `coordinate` shape:

```
\pgfdeclareshape{coordinate}
{
  \savedanchor\centerpoint{%
    \pgf@x=.5\wd\pgfnodeparttextbox%
    \pgf@y=.5\ht\pgfnodeparttextbox%
    \advance\pgf@y by -.5\dp\pgfnodeparttextbox%
  }
  \anchor{center}{\centerpoint}
  \anchorborder{\centerpoint}
}
```

The special commands are explained next. In the examples given for the special commands a new shape will be constructed, which we might call `simple rectangle`. It should behave like the normal rectangle shape, only without bothering about the fine details like inner and outer separations. The skeleton for the shape is the following.

```
\pgfdeclareshape{simple rectangle}{
  ...
}
```

`\nodeparts{`⟨*list of node parts*⟩`}`

This command declares which parts make up nodes of this shape. A *node part* is a (possibly empty) text label that is drawn when a node of the shape is created.

By default, a shape has just one node part called `text`. However, there can be several node parts. For example, the `circle split` shape has two parts: the `text` part, which shows that the upper text, and a `lower` part, which shows the lower text. For the `circle split` shape the `\nodeparts` command was called with the argument `{text,lower}`.

When a multipart node is created, the text labels are drawn in the sequences listed in the ⟨*list of node parts*⟩. For each node part, you must have declared one anchor and the TEX-box of the part is placed at this anchor. For a node part called `XYZ` the TEX-box `\pgfnodepartXYZbox` is placed at anchor `XYZ`.

`\savedanchor{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command declares a saved anchor. The argument ⟨*command*⟩ should be a TEX macro name like `\centerpoint`.

The ⟨*code*⟩ will be executed each time `\pgfnode` (or `\pgfmultipartnode`) is called to create a node of the shape ⟨*shape name*⟩. When the ⟨*code*⟩ is executed, the TEX-boxes of the node parts will contain the text labels of the node. Possibly, these box are void. For example, if there is just a `text` part, the node `\pgfnodeparttextbox` will be set up when the ⟨*code*⟩ is executed.

The ⟨*code*⟩ can use the width, height, and depth of the box(es) to compute the location of the saved anchor. In addition, the ⟨*code*⟩ can take into account the values of dimensions like `\pgfshapeminwidth` or `\pgfshapeinnerxsep`. Furthermore, the ⟨*code*⟩ can take into consideration the values of any further shape-specific variables that are set at the moment when `\pgfnode` is called.

The net effect of the ⟨*code*⟩ should be to set the two TEX dimensions `\pgf@x` and `\pgf@y`. One way to achieve this is to say `\pgfpoint{`⟨*x value*⟩`}{`⟨*y value*⟩`}` at the end of the ⟨*code*⟩, but you can also just set these variables. The values that `\pgf@x` and `\pgf@y` have after the code has been executed, let us call them x and y, will be recorded and stored together with the node that is created by the command `\pgfnode`.

The macro ⟨*command*⟩ is defined to be `\pgfpoint{`x`}{`y`}`. However, the ⟨*command*⟩ is only locally defined while anchor positions are being computed. Thus, it is possible to use very simple names for ⟨*command*⟩, like `\center` or `\a`, without causing a name-clash. (To be precise, very simple ⟨*command*⟩ names will clash with existing names, but only locally inside the computation of anchor positions; and we do not need the normal `\center` command during these computations.)

For our `simple rectangle` shape, we will need only one saved anchor: The upper right corner. The lower left corner could either be the origin or the "mirrored" upper right corner, depending on whether we want the text label to have its lower left corner at the origin or whether the text label should be centered on the origin. Either will be fine, for the final shape this will make no difference since the shape will be shifted anyway. So, let us assume that the text label is centered on the origin (this will be specified later on using the `text` anchor). We get the following code for the upper right corner:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.5\ht\pgfnodeparttextbox % height of the box, ignoring the depth
  \pgf@x=.5\wd\pgfnodeparttextbox % width of the box
}
```

If we wanted to take, say, the `\pgfshapeminwidth` into account, we could use the following code:

```
\savedanchor{\upperrightcorner}{
  \pgf@y=.\ht\pgfnodeparttextbox % height of the box
  \pgf@x=.\wd\pgfnodeparttextbox % width of the box
  \setlength{\pgf@xa}{\pgfshapeminwidth}
  \ifdim\pgf@x<.5\pgf@xa
    \pgf@x=.5\pgf@xa
  \fi
}
```

Note that we could not have written `.5\pgfshapeminwidth` since the minimum width is stored in a "plain text macro", not as a real dimension. So if `\pgfshapeminwidth` depth were 2cm, writing `.5\pgfshapeminwidth` would yield the same as `.52cm`.

In the "real" `rectangle` shape the code is somewhat more complex, but you get the basic idea.

`\saveddimen{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command is similar to `\savedanchor`, only instead of setting ⟨*command*⟩ to `\pgfpoint{`x`}{`y`}`, the ⟨*command*⟩ is set just to (the value of) x.

In the `simple rectangle` shape we might use a saved dimension to store the depth of the shape box.

```
\saveddimen{\depth}{
  \pgf@x=\dp\pgfnodeparttextbox
}
```

`\savedmacro{`⟨*command*⟩`}{`⟨*code*⟩`}`

This command is similar to `\saveddimen`, only at some point in ⟨*code*⟩, ⟨*command*⟩ should be defined appropriately, (this could be a value, or some text).

In the `regular polygon` shape, a saved macro is used to store the number of sides of the polygon.

```
\savedmacro{\sides}{\let\sides\pgfpolygonsides}
```

`\anchor{`⟨*name*⟩`}{`⟨*code*⟩`}`

This command declares an anchor named ⟨*name*⟩. Unlike for saved anchors, the ⟨*code*⟩ will not be executed each time a node is declared. Rather, the ⟨*code*⟩ is only executed when the anchor is specifically requested; either for anchoring the node during its creation or as a position in the shape referenced later on.

The ⟨*name*⟩ is a quite arbitrary string that is not "passed down" to the system level. Thus, names like `south` or `1` or `::` would all be fine.

A saved anchor is not automatically also a normal anchor. If you wish to give the users access to a saved anchor you must declare a normal anchor that just returns the position of the saved anchor.

When the ⟨*code*⟩ is executed, all saved anchor macros will be defined. Thus, you can reference them in your ⟨*code*⟩. The effect of the ⟨*code*⟩ should be to set the values of `\pgf@x` and `\pgf@y` to the coordinates of the anchor.

Let us consider some example for the `simple rectangle` shape. First, we would like to make the upper right corner publicly available, for example as `north east`:

```
\anchor{north east}{\upperrightcorner}
```

The `\upperrightcorner` macro will set `\pgf@x` and `\pgf@y` to the coordinates of the upper right corner. Thus, `\pgf@x` and `\pgf@y` will have exactly the right values at the end of the anchor's code.

Next, let us define a `north west` anchor. For this anchor, we can negate the `\pgf@x` variable:

```
\anchor{north west}{
  \upperrightcorner
  \pgf@x=-\pgf@x
}
```

Finally, it is a good idea to always define a `center` anchor, which will be the default location for a shape.

```
\anchor{center}{\pgfpointorigin}
```

You might wonder whether we should not take into consideration that the node is not placed at the origin, but has been shifted somewhere. However, the anchor positions are always specified in the shape's "private" coordinate system. The "outer" transformation that has been applied to the shape upon its creation is applied automatically to the coordinates returned by the anchor's ⟨*code*⟩.

Our `simple rectangle` only has one text label (node part) called `text`. This is the default situation, so we do not need to do anything. For the `text` node part we must set up a `text` anchor. Upon creation of a node, this anchor will be made to coincide with the left endpoint of the baseline of the text label (within the private coordinate system of the shape). By default, the `text` anchor is at the origin, but you may change this. For example, we would say

```
\anchor{text}{%
  \upperrightcorner%
  \pgf@x=-\pgf@x%
  \pgf@y=-\pgf@y%
}
```

to center the text label on the origin in the shape coordinate space. Note that we could *not* have written the following:

```
\anchor{text}{\pgfpoint{-.5\wd\pgfnodeparttextbox}{-.5\ht\pgfnodeparttextbox}}
```

Do you see why this is wrong? The problem is that the box `\pgfnodeparttextbox` will most likely not have the correct size when the anchor is computed. After all, the anchor position might be recomputed at a time when several other nodes have been created.

If a shape has several node parts, we would have to define an anchor for each part.

`\deferredanchor{`⟨*name*⟩`}{`⟨*code*⟩`}`

This command declares an anchor named ⟨*name*⟩. It works like `\anchor`. However, unlike for anchors declared by `\anchor`, ⟨*name*⟩ will *not* be expanded during the shape declaration (i.e. not during `\pgfdeclareshape`). Rather, the ⟨*name*⟩ is expanded when the *node* is actually used (with `\pgfnode` or more likely with `\node`). This may be useful if the anchor name is context dependent (depending, for example, on the value of a key).

```
\makeatletter
\def\foo{foo}
\pgfdeclareshape{simple shape}{%
  \savedanchor{\center}{%
    \pgfpointorigin}
  \anchor{center}{\center}
  \savedanchor{\anchorfoo}{%
    \pgf@x=1cm
    \pgf@y=0cm}
  \deferredanchor{anchor \foo}{\anchorfoo}}

\begin{tikzpicture}
  \node[simple shape] (Test1) at (0,0) {};
  \fill (Test1.anchor foo) circle [radius=2pt] node[below] {anchor foo anchor};
  %
  \def\foo{bar}
  \node[simple shape] (Test2) at (2,2) {};
  \fill (Test2.anchor bar) circle [radius=2pt] node[below] {anchor bar anchor};
\end{tikzpicture}
```

`\anchorborder{`⟨*code*⟩`}`

A *border anchor* is an anchor point on the border of the shape. What exactly is considered as the "border" of the shape depends on the shape.

When the user requests a point on the border of the shape using the `\pgfpointshapeborder` command, the ⟨*code*⟩ will be executed to discern this point. When the execution of the ⟨*code*⟩ starts, the dimensions `\pgf@x` and `\pgf@y` will have been set to a location p in the shape's coordinate system, and relative to the anchor `center`. Note that `\pgfpointshapeborder` will produce an error if the shape does not contain the `center` anchor.

It is now the job of the ⟨*code*⟩ to set up `\pgf@x` and `\pgf@y` such that they specify the point on the shape's border that lies on a straight line from the shape's center to the point p. Usually, this is a somewhat complicated computation, involving many case distinctions and some basic math. Note that the output coordinates must be returned in the shape's coordinate system, *no longer* relative to the `center` anchor. While these different points of reference are only noticeable if the `center` anchor is not at the origin of the shape's coordinate system, it implies that "doing nothing" as a border anchor, i.e., returning the point that was fed to `\pgfpointshapeborder` requires adding the `center` anchor to the input coordinates.

For our `simple rectangle` we must compute a point on the border of a rectangle whose one corner is the origin (ignoring the depth for simplicity) and whose other corner is `\upperrightcorner`. The following code might be used:

```
\anchorborder{%
  % Call a function that computes a border point. Since this
  % function will modify dimensions like \pgf@x, we must move them to
  % other dimensions.
  \@tempdima=\pgf@x
  \@tempdimb=\pgf@y
  \pgfpointborderrectangle{\pgfpoint{\@tempdima}{\@tempdimb}}{\upperrightcorner}
}
```

\backgroundpath{⟨*code*⟩}

This command specifies the path that "makes up" the background of the shape. Note that the shape cannot prescribe what is going to happen with the path: It might be drawn, shaded, filled, or even thrown away. If you want to specify that something should "always" happen when this shape is drawn (for example, if the shape is a stop-sign, we *always* want it to be filled with a red color), you can use commands like `\beforebackgroundpath`, explained below.

When the ⟨*code*⟩ is executed, all saved anchors will be in effect. The ⟨*code*⟩ should contain path construction commands.

For our `simple rectangle`, the following code might be used:

```
\backgroundpath{
  \pgfpathrectanglecorners
    {\upperrightcorner}
    {\pgfpointscale{-1}{\upperrightcorner}}
}
```

As the name suggests, the background path is used "behind" the text labels. Thus, this path is used first, then the text labels are drawn, possibly obscuring part of the path.

\foregroundpath{⟨*code*⟩}

This command works like `\backgroundpath`, only it is invoked after the text labels have been drawn. This means that this path can possibly obscure (part of) the text labels.

\behindbackgroundpath{⟨*code*⟩}

Unlike the previous two commands, ⟨*code*⟩ should not only construct a path, it should also use this path in whatever way is appropriate. For example, the ⟨*code*⟩ might fill some area with a uniform color.

Whatever the ⟨*code*⟩ does, it does it first. This means that any drawing done by ⟨*code*⟩ will be even behind the background path.

Note that the ⟨*code*⟩ is protected with a `{pgfscope}`.

\beforebackgroundpath{⟨*code*⟩}

This command works like `\behindbackgroundpath`, only the ⟨*code*⟩ is executed after the background path has been used, but before the texts label are drawn.

\behindforegroundpath{⟨*code*⟩}

The ⟨*code*⟩ is executed after the text labels have been drawn, but before the foreground path is used.

\beforeforegroundpath{⟨*code*⟩}

This ⟨*code*⟩ is executed at the very end.

\inheritsavedanchors[from={⟨*another shape name*⟩}]

This command allows you to inherit the code for saved anchors from ⟨*another shape name*⟩. The idea is that if you wish to create a new shape that is just a small modification of a another shape, you can recycle the code used for ⟨*another shape name*⟩.

The effect of this command is the same as if you had called `\savedanchor` and `\saveddimen` for each saved anchor or saved dimension declared in ⟨*another shape name*⟩. Thus, it is not possible to "selectively" inherit only some saved anchors, you always have to inherit all saved anchors from another shape. However, you can inherit the saved anchors of more than one shape by calling this command several times.

`\inheritbehindbackgroundpath[from={`⟨*another shape name*⟩`}]`

This command can be used to inherit the code used for the drawings behind the background path from ⟨*another shape name*⟩.

`\inheritbackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the background path code from ⟨*another shape name*⟩.

`\inheritbeforebackgroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before background path code from ⟨*another shape name*⟩.

`\inheritbehindforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the behind foreground path code from ⟨*another shape name*⟩.

`\inheritforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the foreground path code from ⟨*another shape name*⟩.

`\inheritbeforeforegroundpath[from={`⟨*another shape name*⟩`}]`

Inherits the before foreground path code from ⟨*another shape name*⟩.

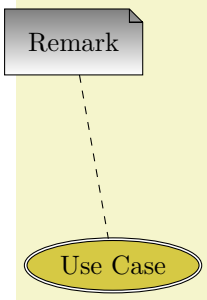
`\inheritanchor[from={`⟨*another shape name*⟩`}]{`⟨*name*⟩`}`

Inherits the code of one specific anchor named ⟨*name*⟩ from ⟨*another shape name*⟩. Thus, unlike saved anchors, which must be inherited collectively, normal anchors can and must be inherited individually.

`\inheritanchorborder[from={`⟨*another shape name*⟩`}]`

Inherits the border anchor code from ⟨*another shape name*⟩.

The following example shows how a shape can be defined that relies heavily on inheritance:




```
\usetikzlibrary {shapes.geometric}
\pgfdeclareshape{document}{
  \inheritsavedanchors[from=rectangle] % this is nearly a rectangle
  \inheritanchorborder[from=rectangle]
  \inheritanchor[from=rectangle]{center}
  \inheritanchor[from=rectangle]{north}
  \inheritanchor[from=rectangle]{south}
  \inheritanchor[from=rectangle]{west}
  \inheritanchor[from=rectangle]{east}
  % ... and possibly more
  \backgroundpath{% this is new
    % store lower right in xa/ya and upper right in xb/yb
    \southwest \pgf@xa=\pgf@x \pgf@ya=\pgf@y
    \northeast \pgf@xb=\pgf@x \pgf@yb=\pgf@y
    % compute corner of ``flipped page''
    \pgf@xc=\pgf@xb \advance\pgf@xc by-5pt % this should be a parameter
    \pgf@yc=\pgf@yb \advance\pgf@yc by-5pt
    % construct main path
    \pgfpathmoveto{\pgfpoint{\pgf@xa}{\pgf@ya}}
    \pgfpathlineto{\pgfpoint{\pgf@xa}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@ya}}
    \pgfpathclose
    % add little corner
    \pgfpathmoveto{\pgfpoint{\pgf@xc}{\pgf@yb}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xb}{\pgf@yc}}
    \pgfpathlineto{\pgfpoint{\pgf@xc}{\pgf@yc}}
 }
}\hskip-1.2cm
\begin{tikzpicture}
  \node[shade,draw,shape=document,inner sep=2ex] (x) {Remark};
  \node[fill=yellow!80!black,draw,ellipse,double]
    at ([shift=(-80:3cm)]x) (y) {Use Case};

  \draw[dashed] (x) -- (y);
\end{tikzpicture}
```

107 Matrices

```
\usepgfmodule{matrix}  % LaTeX and plain TeX when using pure PGF
\usepgfmodule[matrix]  % ConTeXt when using pure PGF
```

The present section documents the commands of this module.

107.1 Overview

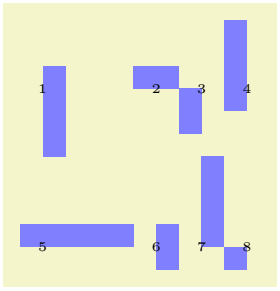
Matrices are a mechanism for aligning several so-called cell pictures horizontally and vertically. The resulting alignment is placed in a normal node and the command for creating matrices, `\pgfmatrix`, takes options very similar to the `\pgfnode` command.

In the following, the basic idea behind the alignment mechanism is explained first. Then the command `\pgfmatrix` is explained. At the end of the section, additional ways of modifying the width of columns and rows are discussed.

107.2 Cell Pictures and Their Alignment

A matrix consists of rows of *cells*. Cells are separated using the special command `\pgfmatrixnextcell`, rows are ended using the command `\pgfmatrixendrow` (the command `\\` is set up to mean the same as `\pgfmatrixendrow` by default). Each cell contains a *cell picture*, although cell pictures are not complete pictures as they lack layers. However, each cell picture has its own bounding box like a normal picture does. These bounding boxes are important for the alignment as explained in the following.

Each cell picture will have an origin somewhere in the picture (or even outside the picture). The position of these origins are important for the alignment: On each row the origins will be on the same horizontal line and for each column the origins will also be on the same vertical line. These two requirements mean that the cell pictures may need to be shifted around so that the origins wind up on the same lines. The top of a row is given by the top of the cell picture whose bounding box's maximum y-position is largest. Similarly, the bottom of a row is given by the bottom of the cell picture whose bounding box's minimum y-position is the most negative. Similarly, the left end of a column is given by the left end of the cell whose bounding box's x-position is the most negative; and similarly for the right end of a column.




```
\begin{tikzpicture}[x=3mm,y=3mm,fill=blue!50]
  \def\atorig#1{\node[black] at (0,0) {\tiny #1};}

  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{}
    {
      \fill (0,-3)  rectangle (1,1);\atorig1 \pgfmatrixnextcell
      \fill (-1,0)  rectangle (1,1);\atorig2 \pgfmatrixnextcell
      \fill (-1,-2) rectangle (0,0);\atorig3 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,3);\atorig4 \\
      \fill (-1,0)  rectangle (4,1);\atorig5 \pgfmatrixnextcell
      \fill (0,-1)  rectangle (1,1);\atorig6 \pgfmatrixnextcell
      \fill (0,0)   rectangle (1,4);\atorig7 \pgfmatrixnextcell
      \fill (-1,-1) rectangle (0,0);\atorig8 \\
    }
\end{tikzpicture}
```

107.3 The Matrix Command

All matrices are typeset using the following command:

`\pgfmatrix`{⟨*shape*⟩}{⟨*anchor*⟩}{⟨*name*⟩}{⟨*usage*⟩}{⟨*shift*⟩}{⟨*pre-code*⟩}{⟨*matrix cells*⟩}

This command creates a node that contains a matrix. The name of the node is ⟨*name*⟩, its shape is ⟨*shape*⟩ and the node is anchored at ⟨*anchor*⟩.

The ⟨*matrix cell*⟩ parameter contains the cells of the matrix. In each cell drawing commands may be given, which create a so-called cell picture. For each cell picture a bounding box is computed and the cells are aligned according to the rules outlined in the previous section.

The resulting matrix is used as the `text` box of the node. As for a normal node, the ⟨*usage*⟩ commands are applied, so that the path(s) of the resulting node is (are) stroked or filled or whatever.

Specifying the cells and rows. Even though this command uses `\halign` internally, there are two special rules for indicating cells:

1. Cells in the same row must be separated using the macro `\pgfmatrixnextcell` rather than `&`. Using `&` will result in an error message.

   However, you can make `&` an active character and have it expand to `\pgfmatrixnextcell`. This way, it will "look" as if `&` is used.

2. Rows are ended using the command `\pgfmatrixendrow`, but `\\` is set up to mean the same by default. However, some environments like `{minipage}` redefine `\\`, so it is good to have `\pgfmatrixendrow` as a "fallback".

3. Every row *including the last row* must be ended using the command `\\` or `\pgfmatrixendrow`.

Both `\pgfmatrixnextcell` and `\pgfmatrixendrow` (and, thus, also `\\`) take an optional argument as explained in the Section 107.4

a b
c d

```
\begin{tikzpicture}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{}
    {
      \node {a}; \pgfmatrixnextcell \node {b}; \pgfmatrixendrow
      \node {c}; \pgfmatrixnextcell \node {d}; \pgfmatrixendrow
    }
\end{tikzpicture}
```

Anchoring matrices at nodes inside the matrix. The parameter ⟨*shift*⟩ is an additional negative shift for the node. Normally, such a shift could be given beforehand (that is, the shift could be preapplied to the current transformation matrix). However, when ⟨*shift*⟩ is evaluated, you can refer to *temporary* positions of nodes inside the matrix. In detail, the following happens: When the matrix has been typeset, all nodes in the matrix temporarily get assigned their positions in the matrix box. The origin of this coordinate system is at the left baseline end of the matrix box, which corresponds to the `text` anchor. The position ⟨*shift*⟩ is then interpreted inside this coordinate system and then used for shifting.

This allows you to use the parameter ⟨*shift*⟩ in the following way: If you use `text` as the ⟨*anchor*⟩ and specify `\pgfpointanchor{inner node}{some anchor}` for the parameter ⟨*shift*⟩, where `inner node` is a node that is created in the matrix, then the whole matrix will be shifted such that `inner node.some anchor` lies at the origin of the whole picture.

Rotations and scaling. The matrix node is never rotated or scaled, because the current coordinate transformation matrix is reset (except for the translational part) at the beginning of `\pgfmatrix`. This is intentional and will not change in the future. If you need to rotate or scale the matrix, you must install an appropriate canvas transformation yourself.

However, nodes and stuff inside the cell pictures can be rotated and scaled normally.

Callbacks. At the beginning and at the end of each cell the special macros `\pgfmatrixbegincode`, `\pgfmatrixendcode` and possibly `\pgfmatrixemptycode` are called. The effect is explained in Section 107.5.

Executing extra code. The parameter ⟨*pre-code*⟩ is executed at the beginning of the outermost TEX-group enclosing the matrix node. It is inside this TEX-group, but outside the matrix itself. It can be used for different purposes:

1. It can be used to simplify the next cell macro. For example, saying `\let\&=\pgfmatrixnextcell` allows you to use `\&` instead of `\pgfmatrixnextcell`. You can also set the catcode of `&` to active.
2. It can be used to issue an `\aftergroup` command. This allows you to regain control after the `\pgfmatrix` command. (If you do not know the `\aftergroup` command, you are probably blessed with a simple and happy life.)

Special considerations concerning macro expansion. As said before, the matrix is typeset using `\halign` internally. This command does a lot of strange and magic things like expanding the first macro of every cell in a most unusual manner. Here are some effects you may wish to be aware of:

- It is not necessary to actually mention `\pgfmatrixnextcell` or `\pgfmatrixendrow` inside the ⟨*matrix cells*⟩. It suffices that the macros inside ⟨*matrix cells*⟩ expand to these macros sooner or later.
- In particular, you can define clever macros that insert columns and rows as needed for special effects.

107.4 Row and Column Spacing

It is possible to control the space between columns and rows rather detailedly. Two commands are important for the row spacing and two commands for the column spacing.

`\pgfsetmatrixcolumnsep{`⟨*sep list*⟩`}`

This macro sets the default separation list for columns. The details of the format of this list are explained in the description of the next command.

`\pgfmatrixnextcell[`⟨*additional sep list*⟩`]`

This command has two purposes: First, it is used to separate cells. Second, by providing the optional argument ⟨*additional sep list*⟩ you can modify the spacing between the columns that are separated by this command.

The optional ⟨*additional sep list*⟩ may only be provided when the `\pgfmatrixnextcell` command starts a new column. Normally, this will only be the case in the first row, but sometimes a later row has more elements than the first row. In this case, the `\pgfmatrixnextcell` commands that start the new columns in the later row may also have the optional argument. Once a column has been started, subsequent uses of this optional argument for the column have no effect.

To determine the space between the two columns that are separated by `\pgfmatrixnextcell`, the following algorithm is executed:

1. Both the default separation list (as set up by `\pgfsetmatrixcolumnsep`) and the ⟨*additional sep list*⟩ are processed, in this order. If the ⟨*additional sep list*⟩ argument is missing, only the default separation list is processed.
2. Both lists may contain dimensions, separated by commas, as well as occurrences of the keywords `between origins` and `between borders`.
3. All dimensions occurring in either list are added together to arrive at a dimension d.
4. The last occurrence of either of the keywords is located. If neither keyword is present, we proceed as if `between borders` were present.

At the end of the algorithm, a dimension d has been computed and one of the two *modes* `between borders` and `between origins` has been determined. Depending on which mode has been determined, the following happens:

- For the `between borders` mode, an additional horizontal space of d is added between the two columns. Note that d may be negative.
- For the `between origins` mode, the spacing between the two columns is computed differently: Recall that the origins of the cell pictures in both pictures lie on two vertical lines. The spacing between the two columns is set up such that the horizontal distance between these two lines is exactly d.

  This mode may only be used between columns *already introduced in the first row.*

All of the above rules boil down to the following effects:

- A default spacing between columns should be set up using `\pgfsetmatrixcolumnsep`. For example, you might say `\pgfsetmatrixcolumnsep{5pt}` to have columns spaced apart by `5pt`. You could say

  ```
  \pgfsetmatrixcolumnsep{1cm,between origins}
  ```

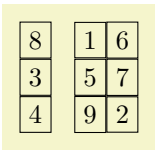
  to specify that horizontal space between the origins of cell pictures in adjacent columns should be 1cm by default – regardless of the actual size of the cell pictures.

- You can now use the optional argument of `\pgfmatrixnextcell` to locally overrule the spacing between two columns. By saying `\pgfmatrixnextcell[5pt]` you *add* 5pt to the space between of the two columns, regardless of the mode.

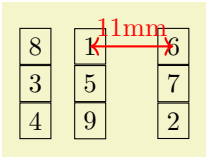
  You can also (locally) change the spacing mode for these two columns. For example, even if the normal spacing mode is `between origins`, you can say

  ```
  \pgfmatrixnextcell[5pt,between borders]
  ```

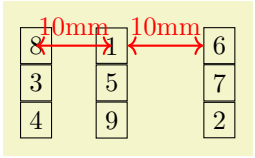
  to locally change the mode for these columns to `between borders`.




```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {8}; \&[2mm] \node{1}; \&[-1mm] \node {6}; \\
    \node {3}; \&      \node{5}; \&       \node {7}; \\
    \node {4}; \&      \node{9}; \&       \node {2}; \\
  }
\end{tikzpicture}
```




```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1mm}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {8}; \&[2mm] \node(a){1}; \&[1cm,between origins] \node(b){6}; \\
    \node {3}; \&      \node   {5}; \&                     \node   {7}; \\
    \node {4}; \&      \node   {9}; \&                     \node   {2}; \\
  }
  \draw [<->,red,thick,every node/.style=] (a.center) -- (b.center)
        node [above,midway] {11mm};
\end{tikzpicture}
```




```
\begin{tikzpicture}[every node/.style=draw]
  \pgfsetmatrixcolumnsep{1cm,between origins}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node (a) {8}; \& \node (b) {1}; \&[between borders] \node (c) {6}; \\
    \node     {3}; \& \node     {5}; \&                  \node     {7}; \\
    \node     {4}; \& \node     {9}; \&                  \node     {2}; \\
  }
  \begin{scope}[every node/.style=]
    \draw [<->,red,thick] (a.center) -- (b.center) node [above,midway] {10mm};
    \draw [<->,red,thick] (b.east) -- (c.west) node [above,midway]
    {10mm};
  \end{scope}
\end{tikzpicture}
```

The mechanism for the between-row-spacing is the same, only the commands are called differently.

`\pgfsetmatrixrowsep{`⟨*sep list*⟩`}`

This macro sets the default separation list for rows.

`\pgfmatrixendrow[`⟨*additional sep list*⟩`]`

This command ends a line. The optional ⟨*additional sep list*⟩ is used to determine the spacing between the row being ended and the next row. The modes and the computation of d is done in the same way as for columns. For the last row the optional argument has no effect.

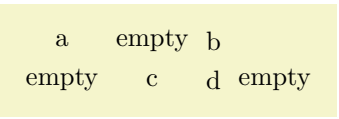
Inside matrices (and only there) the command `\\` is set up to mean the same as this command.

107.5 Callbacks

There are three macros that get called at the beginning and end of cells. By redefining these macros, which are empty by default, you can change the appearance of cells in a very general manner.

\pgfmatrixemptycode

This macro is executed for empty cells. This means that PGF uses some macro magic to determine whether a cell is empty (it immediately ends with \pgfmatrixemptycode or \pgfmatrixendrow) and, if so, put this macro inside the cell.

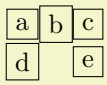



```
\begin{tikzpicture}
  \def\pgfmatrixemptycode{\node{empty};}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    \node {a}; \&           \& \node {b}; \\
               \& \node{c}; \& \node {d}; \& \\
  }
\end{tikzpicture}
```

As can be seen, the macro is not executed for empty cells at the end of row when columns are added only later on.

\pgfmatrixbegincode

This macro is executed at the beginning of non-empty cells. Correspondingly, \pgfmatrixendcode is added at the end of every non-empty cell.




```
\begin{tikzpicture}
  \def\pgfmatrixbegincode{\node[draw]\bgroup}
  \def\pgfmatrixendcode{\egroup;}
  \pgfmatrix{rectangle}{center}{mymatrix}
    {\pgfusepath{}}{\pgfpointorigin}{\let\&=\pgfmatrixnextcell}
  {
    a \& b \& c \\
    d \&   \& e \\
  }
\end{tikzpicture}
```

Note that between \pgfmatrixbegincode and \pgfmatrixendcode there will *not* only be the contents of the cell. Rather, PGF will add some (invisible) commands for book-keeping purposes that involve \let and \gdef. In particular, it is not a good idea to have \pgfmatrixbegincode end with \csname and \pgfmatrixendcode start with \endcsname.

\pgfmatrixendcode

See the explanation above.

The following two counters allow you to access the current row and current column in a callback:

\pgfmatrixcurrentrow

This counter stores the current row of the current cell of the matrix. Do not even think about changing this counter.

\pgfmatrixcurrentcolumn

This counter stores the current column of the current cell of the matrix.

108 Coordinate, Canvas, and Nonlinear Transformations

108.1 Overview

PGF offers different ways of scaling, shifting, and rotating (these operations are generally known as *transformations*) graphics: You can apply *coordinate transformations* to all coordinates, you can apply *canvas transformations* to the canvas on which you draw, and you can apply *additional nonlinear transformations*. (The names "coordinate" and "canvas" transformations are not standard, I introduce them only for the purposes of this manual.)

The differences are the following:

- As the name "coordinate transformation" suggests, coordinate transformations apply only to coordinates. For example, when you specify a coordinate like `\pgfpoint{1cm}{2cm}` and you wish to "use" this coordinate – for example as an argument to a `\pgfpathmoveto` command – then the coordinate transformation matrix is applied to the coordinate, resulting in a new coordinate. Continuing the example, if the current coordinate transformation is "scale by a factor of two", the coordinate `\pgfpoint{1cm}{2cm}` actually designates the point $(2\text{cm}, 4\text{cm})$.

  Note that coordinate transformations apply *only* to coordinates. They do not apply to, say, line width or shadings or text.

- The effect of a "canvas transformation" like "scale by a factor of two" can be imagined as follows: You first draw your picture on a "rubber canvas" normally. Then, once you are done, the whole canvas is transformed, in this case stretched by a factor of two. In the resulting image *everything* will be larger: Text, lines, coordinates, and shadings.

- Nonlinear transformations are a special form of coordinate transformations that are, as the name suggests, not linear. The support for nonlinear transformations is quite different from the support for linear coordinate transformations, the main reason being speed: While linear coordinate transformations can be applied very quickly (PGF does so almost constantly), nonlinear transformations are much harder to apply and also to use. For this reason, nonlinear transformations are implemented in a special module `nonlineartransformations` that has to be loaded explicitly. By default, they are not available.

In many cases, it is preferable that you use coordinate transformations and not canvas transformations. When canvas transformations are used, PGF looses track of the coordinates of nodes and shapes. Also, canvas transformations often cause undesirable effects like changing text size. For these reasons, PGF makes it easy to setup the coordinate transformation, but a bit harder to change the canvas transformation. Because of the speed penalties caused by nonlinear transformations, they are even harder to set up.

108.2 Coordinate Transformations

108.2.1 How PGF Keeps Track of the Coordinate Transformation Matrix

PGF has an internal coordinate transformation matrix. This matrix is applied to coordinates "in certain situations". This means that the matrix is not always applied to every coordinate "no matter what". Rather, PGF tries to be reasonably smart at when and how this matrix should be applied. The most prominent examples are the path construction commands, which apply the coordinate transformation matrix to their inputs.

The coordinate transformation matrix consists of four numbers a, b, c, and d, and two dimensions s and t. When the coordinate transformation matrix is applied to a coordinate (x, y), the new coordinate $(ax + cy + s, bx + dy + t)$ results. For more details on how transformation matrices work in general, please see, for example, the PDF or PostScript reference or a textbook on computer graphics.

The coordinate transformation matrix is equal to the identity matrix at the beginning. More precisely, $a = 1$, $b = 0$, $c = 0$, $d = 1$, $s = 0\text{pt}$, and $t = 0\text{pt}$.

The different coordinate transformation commands will modify the matrix by concatenating it with another transformation matrix. This way the effect of applying several transformation commands will *accumulate*.

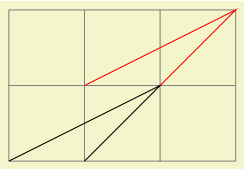
The coordinate transformation matrix is local to the current TEX group (unlike the canvas transformation matrix, which is local to the current `{pgfscope}`). Thus, the effect of adding a coordinate transformation to the coordinate transformation matrix will last only till the end of the current TEX group.

108.2.2 Commands for Relative Coordinate Transformations

The following commands add a basic coordinate transformation to the current coordinate transformation matrix. For all commands, the transformation is applied *in addition* to any previous coordinate transformations.

\pgftransformshift{⟨*point*⟩}

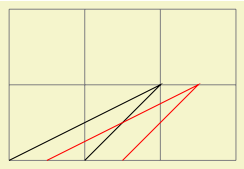
Shifts coordinates by ⟨*point*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformshift{\pgfpoint{1cm}{1cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformxshift{⟨*dimensions*⟩}

Shifts coordinates by ⟨*dimension*⟩ along the x-axis.



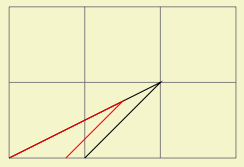
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxshift{.5cm}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyshift{⟨*dimensions*⟩}

Like `\pgftransformxshift`, only for the y-axis.

\pgftransformscale{⟨*factor*⟩}

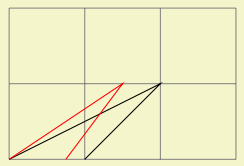
Scales coordinates by ⟨*factor*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformxscale{⟨*factor*⟩}

Scales coordinates by ⟨*factor*⟩ in the x-direction.



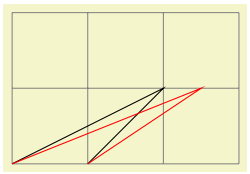
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxscale{.75}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyscale{⟨*factor*⟩}

Like `\pgftransformxscale`, only for the y-axis.

\pgftransformxslant{⟨*factor*⟩}

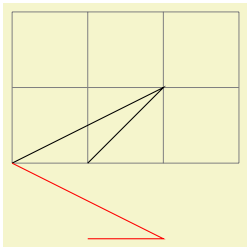
Slants coordinates by ⟨*factor*⟩ in the x-direction. Here, a factor of `1` means 45°.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformxslant{.5}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformyslant{⟨*factor*⟩}

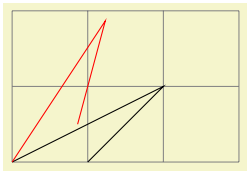
Slants coordinates by ⟨*factor*⟩ in the y-direction.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformyslant{-1}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformrotate{⟨*angles*⟩}

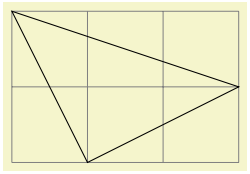
Rotates coordinates counterclockwise by ⟨*angles*⟩ given in degrees.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformrotate{30}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformtriangle{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}

This command transforms the coordinate system in such a way that the triangle given by the points ⟨*a*⟩, ⟨*b*⟩ and ⟨*c*⟩ lies at the coordinates $(0, 0)$, $(1\text{pt}, 0\text{pt})$ and $(0\text{pt}, 1\text{pt})$.

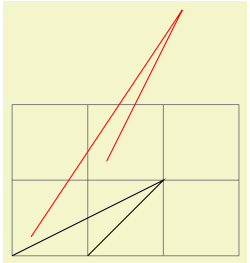


```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformtriangle
  {\pgfpoint{1cm}{0cm}}
  {\pgfpoint{0cm}{2cm}}
  {\pgfpoint{3cm}{1cm}}

  \draw (0,0) -- (1pt,0pt) -- (0pt,1pt) -- cycle;
\end{tikzpicture}
```

\pgftransformcm{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*point*⟩}

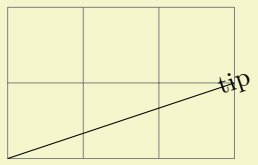
Applies the transformation matrix given by a, b, c, and d and the shift ⟨*point*⟩ to coordinates (in addition to any previous transformations already in force).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformcm{1}{1}{0}{1}{\pgfpoint{.25cm}{.25cm}}
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformarrow{⟨*start*⟩}{⟨*end*⟩}

Shifts coordinates to the end of the line going from ⟨*start*⟩ to ⟨*end*⟩ with the correct rotation.

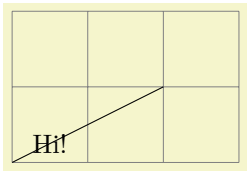



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (3,1);
  \pgftransformarrow{\pgfpointorigin}{\pgfpoint{3cm}{1cm}}
  \pgftext{tip}
\end{tikzpicture}
```

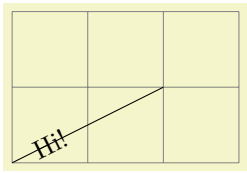
\pgftransformlineattime{⟨*time*⟩}{⟨*start*⟩}{⟨*end*⟩}

Shifts coordinates by a specific point on a line at a specific time. The point by which the coordinate is shifted is calculated by calling `\pgfpointlineattime`, see Section 101.5.2.

In addition to shifting the coordinate, a rotation *may* also be applied. Whether this is the case depends on whether the TEX if `\ifpgfslopedattime` is set to true or not.



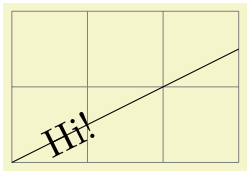

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1);
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```



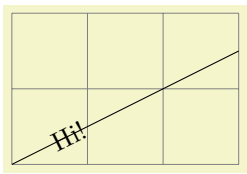

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

If `\ifpgfslopedattime` is true, another TEX `\if` is important: `\ifpgfallowupsidedownattime`. If this is false, PGF will ensure that the rotation is done in such a way that text is never "upside down".

There is another TEX `\if` that influences this command. If you set `\ifpgfresetnontranslationattime` to true, then, between shifting the coordinate and (possibly) rotating/sloping the coordinate, the command `\pgftransformresetnontranslations` is called. See the description of this command for details.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimefalse
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

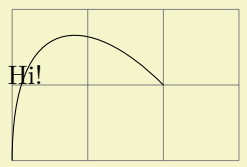



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{1.5}
  \draw      (0,0) -- (2,1);
  \pgfslopedattimetrue
  \pgfresetnontranslationattimetrue
  \pgftransformlineattime{.25}{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

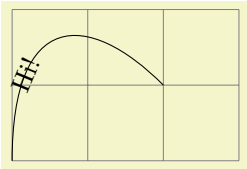
\pgftransformcurveattime{⟨*time*⟩}{⟨*start*⟩}{⟨*first support*⟩}{⟨*second support*⟩}{⟨*end*⟩}

Shifts coordinates by a specific point on a curve at a specific time, see Section 101.5.2 once more.

As for the line-at-time transformation command, `\ifpgfslopedattime` decides whether an additional rotation should be applied. Again, the value of `\ifpgfallowupsidedownattime` is also considered.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```



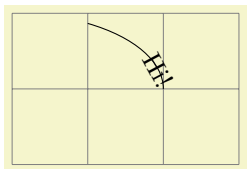

```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \draw      (0,0) .. controls (0,2) and (1,2) .. (2,1);
  \pgfslopedattimetrue
  \pgftransformcurveattime{.25}{\pgfpointorigin}
    {\pgfpoint{0cm}{2cm}}{\pgfpoint{1cm}{2cm}}{\pgfpoint{2cm}{1cm}}
  \pgftext{Hi!}
\end{tikzpicture}
```

The value of `\ifpgfresetnontranslationsattime` is also taken into account.

`\pgftransformarcaxesattime{`⟨*time t*⟩`}{`⟨*center*⟩`}{`⟨*0-degree axis*⟩`}{`⟨*90-degree axis*⟩`}{`⟨*start angle*⟩`}{`⟨*end angle*⟩`}`

Shifts coordinates by a specific point on an arc at a specific time, see Section 101.5.2 once more.

As for the previous commands, `\ifpgfslopedattime` decides whether an additional rotation should be applied and `\ifpgfallowupsidedownattime` is also considered.




```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathmoveto{\pgfpoint{2cm}{1cm}}
  \pgfpatharcaxes{0}{60}{\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}
  \pgfusepath{stroke}
  \pgfslopedattimetrue
  \pgftransformarcaxesattime{.25}
    {\pgfpoint{0cm}{1cm}}
    {\pgfpoint{2cm}{0cm}}{\pgfpoint{0cm}{1cm}}
    {0}{60}
  \pgftext{Hi!}
\end{tikzpicture}
```

The value of `\ifpgfresetnontranslationsattime` is also taken into account.

`\ifpgfslopedattime`

Decides whether the "at time" transformation commands also rotate coordinates or not.

`\ifpgfallowupsidedownattime`

Decides whether the "at time" transformation commands should allow the rotation be done in such a way that "upside-down text" can result.

`\ifpgfresetnontranslationsattime`

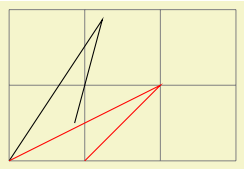
Decides whether the "at time" transformation commands should reset the non-translations between shifting and rotating.

108.2.3 Commands for Absolute Coordinate Transformations

The coordinate transformation commands introduced up to now are always applied in addition to any previous transformations. In contrast, the commands presented in the following can be used to change the transformation matrix "in absolute terms". Note that this is, in general, dangerous and will often produce unexpected effects. You should use these commands only if you really know what you are doing.

`\pgftransformreset`

Resets the coordinate transformation matrix to the identity matrix. Thus, once this command is given no transformations are applied till the end of the scope.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransformreset
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

\pgftransformresetnontranslations

This command sets the a, b, c, and d part of the coordinate transformation matrix to $a = 1$, $b = 0$, $c = 0$, and $d = 1$. However, the current shifting of the matrix is not modified.

The effect of this command is that any rotation/scaling/slanting is undone in the current TEX group, but the origin is not "moved back".

This command is mostly useful directly before a \pgftext command to ensure that the text is not scaled or rotated.

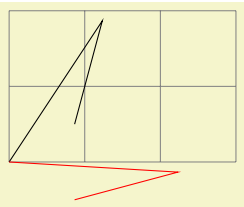



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformscale{2}
  \pgftransformrotate{30}
  \pgftransformxshift{1cm}
  {\color{red}\pgftext{rotated}}
  \pgftransformresetnontranslations
  \pgftext{shifted only}
\end{tikzpicture}
```

\pgftransforminvert

Replaces the coordinate transformation matrix by a coordinate transformation matrix that "exactly undoes the original transformation". For example, if the original transformation was "scale by 2 and then shift right by 1cm" the new one is "shift left by 1cm and then scale by 1/2".

This command will produce an error if the determinant of the matrix is too small, that is, if the matrix is near-singular.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformrotate{30}
  \draw      (0,0) -- (2,1) -- (1,0);
  \pgftransforminvert
  \draw[red] (0,0) -- (2,1) -- (1,0);
\end{tikzpicture}
```

108.2.4 Saving and Restoring the Coordinate Transformation Matrix

There are two commands for saving and restoring coordinate transformation matrices.

\pgfgettransform{⟨*macro*⟩}

This command will (locally) define ⟨*macro*⟩ to a representation of the current coordinate transformation matrix. This matrix can later on be reinstalled using \pgfsettransform.

\pgfsettransform{⟨*macro*⟩}

Reinstalls a coordinate transformation matrix that was previously saved using \pgfgettransform.

\pgfgettransformentries{⟨*macro for a*⟩}{⟨*macro for b*⟩}{⟨*macro for c*⟩}{⟨*macro for d*⟩}{⟨*macro for shift x*⟩}{⟨*macro for shift y*⟩}

This command is similar to \pgfgettransform except that it stores the current coordinate transformation matrix in a set of six macros.

The matrix can later on be reinstalled using \pgfsettransformentries. Furthermore, all these macros (or just a few of them) can be used as arguments for \pgftransformcm.

\pgfsettransformentries{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨shiftx⟩}{⟨shifty⟩}

Reinstalls a coordinate transformation matrix that was previously saved using the storage command \pgfgettransformentries. This command can also be used to replace any previously existing coordinate transformation matrix (it is thus equivalent to \pgftransformreset followed by \pgftransformcm).

108.2.5 Applying Coordinate Transformation to Points

\pgfpointtransformed{⟨point⟩}

Applies current transformation matrix to {⟨point⟩} (x, y) and returns a transformed point $(ax + cy + s, bx + dy + t)$. Normally, this is done automatically by commands like \pgfpathlineto or \pgfpathmoveto, but sometimes you may wish to access a transformed point yourself.

108.2.6 Computing Adjustments for Coordinate Transformations

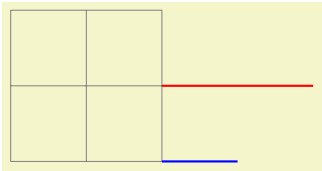
\pgftransformationadjustments

This command computes "adjustments" for the current transformation matrix so that even when you install a transformation matrix that scales everything by a certain factor, you can still draw something of "an absolute size". Suppose for instance that you install a transformation matrix that scales everything by a factor of 4 and you now wish to draw a horizontal line of length 1cm. Then, if you do not reset the transformation matrix, you can draw a line of logical length 2.5mm, which will then get scaled to a line of 1cm. Things get more difficult in case you scale things only, say, vertically. In this case, the adjustment necessary for horizontal lines is different from the one needed for vertical lines.

This function computes two scaling factors, one for horizontal lines and one for vertical lines, and stores them in the following macros:

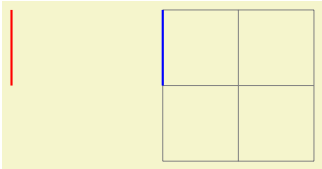
\pgfhorizontaltransformationadjustment

When you scale the length of a horizontal line by this factor in the current transformation, you compensate for the scaling. Formally, it is $1/\|transform(1,0)\|_2$, where $transform$ applies the current transformations matrix to the given number.



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \begin{scope}[xscale=2,thick]
    \draw [red] (1,1) -- ++(1,0);

    \pgftransformationadjustments
    \draw [blue] (1,0) -- ++(\pgfhorizontaltransformationadjustment,0);
  \end{scope}
\end{tikzpicture}
```



```
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);
  \begin{scope}[xscale=2,thick,rotate=90]
    \draw [red] (1,1) -- ++(1,0);

    \pgftransformationadjustments
    \draw [blue] (1,0) -- ++(\pgfhorizontaltransformationadjustment,0);
  \end{scope}
\end{tikzpicture}
```

\pgfverticaltransformationadjustment

$1/\|transform(0,1)\|_2$.

Note that the "right" way to draw a line of absolute length 1cm in a transformed coordinate system is to first compute the start point and to then reset the transformation matrix. The transformation adjustments computed here are important only in situations where you cannot do this, for instance when an `outer xsep` must be set.

108.3 Canvas Transformations

The canvas transformation matrix is not managed by PGF, but by the output format like PDF or PostScript. All that PGF does is to call appropriate low-level `\pgfsys@` commands to change the canvas transformation matrix.

Unlike coordinate transformations, canvas transformations apply to "everything", including images, text, shadings, line thickness, and so on. The idea is that a canvas transformation really stretches and deforms the canvas after the graphic is finished.

Unlike coordinate transformations, canvas transformations are local to the current `{pgfscope}`, not to the current TEX group. This is due to the fact that they are managed by the backend driver, not by TEX or PGF.

Unlike the coordinate transformation matrix, it is not possible to "reset" the canvas transformation matrix. The only way to change it is to concatenate it with another canvas transformation matrix or to end the current `{pgfscope}`.

Unlike coordinate transformations, PGF does not "keep track" of canvas transformations. In particular, it will not be able to correctly save the coordinates of shapes or nodes when a canvas transformation is used.

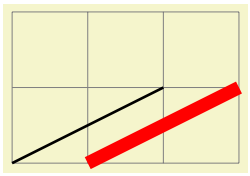
108.3.1 Applying General Canvas Transformations

PGF does not offer many commands for modifying the canvas transformation matrix. Instead, different commands allow you to concatenate the canvas transformation matrix with a coordinate transformation matrix (and there are numerous commands for specifying a coordinate transformation, see the previous section).

`\pgflowlevelsynccm`

This command concatenates the canvas transformation matrix with the current coordinate transformation matrix. Afterward, the coordinate transformation matrix is reset.

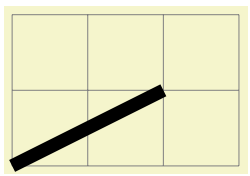
The effect of this command is to "synchronize" the coordinate transformation matrix and the canvas transformation matrix. All transformations that were previously applied by the coordinate transformations matrix are now applied by the canvas transformation matrix.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgftransformscale{5}
  \draw      (0,0) -- (0.4,.2);
  \pgftransformxshift{0.2cm}
  \pgflowlevelsynccm
  \draw[red] (0,0) -- (0.4,.2);
\end{tikzpicture}
```

`\pgflowlevel{`⟨*transformation code*⟩`}`

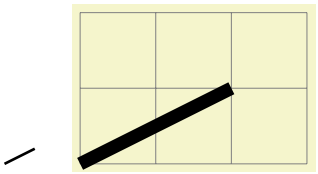
This command concatenates the canvas transformation matrix with the coordinate transformation specified by ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevel{\pgftransformscale{5}}
  \draw      (0,0) -- (0.4,.2);
\end{tikzpicture}
```

`\pgflowlevelobj{`⟨*transformation code*⟩`}{`⟨*code*⟩`}`

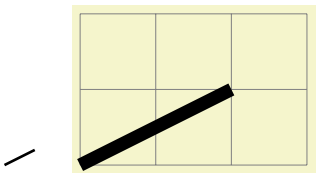
This command creates a local `{pgfscope}`. Inside this scope, `\pgflowlevel` is first called with the argument ⟨*transformation code*⟩, then the ⟨*code*⟩ is inserted.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \pgflowlevelobj{\pgftransformscale{5}}    {\draw (0,0) -- (0.4,.2);}
  \pgflowlevelobj{\pgftransformxshift{-1cm}}{\draw (0,0) -- (0.4,.2);}
\end{tikzpicture}
```

\begin{pgflowlevelscope}{⟨*transformation code*⟩}
  ⟨*environment contents*⟩
\end{pgflowlevelscope}

This environment first surrounds the ⟨*environment contents*⟩ by a `{pgfscope}`. Then it calls `\pgflowlevel` with the argument ⟨*transformation code*⟩.



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfsetlinewidth{1pt}
  \begin{pgflowlevelscope}{\pgftransformscale{5}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
  \begin{pgflowlevelscope}{\pgftransformxshift{-1cm}}
    \draw (0,0) -- (0.4,.2);
  \end{pgflowlevelscope}
\end{tikzpicture}
```

\pgflowlevelscope{⟨*transformation code*⟩}
  ⟨*environment contents*⟩
\endpgflowlevelscope

The plain TEX version of the environment.

\startpgflowlevelscope{⟨*transformation code*⟩}
  ⟨*environment contents*⟩
\stoppgflowlevelscope

The ConTEXt version of the environment.

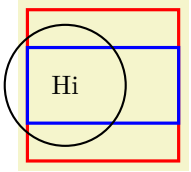
108.3.2 Establishing View Boxes

A *view box* is like a "window" through which you see a graphic. To establish a view box, you specify a rectangle – which is the window – and another rectangle surrounding the to-be-viewed graphic. The graphic will then be rescaled and shifted in such a way that the to-be-viewed rectangle matches the view box's rectangle as well as possible. Note that establishing a view box does, indeed, cause a canvas transformation to be installed.

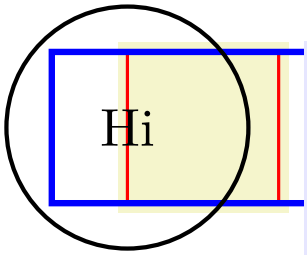
View boxes are only seldom needed in normal graphics. Their main application is with animations since you can *animate* the to-be-viewed rectangle. This makes it easy to create animations in which you zoom in, zoom out, and pan a graphic.

\begin{pgfviewboxscope}{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  ⟨*environment contents*⟩
\end{pgfviewboxscope}

Inside the viewbox scope, the source rectangle (with the two PGF points ll_1 and ur_1 as corners) will be translated and scaled so that it becomes centered on the target rectangle (with the corners ll_2 and ur_2) and will, for `meet` as last parameter, be as large as possible so that it fits inside the target and, for `slice`, be as small as possible so that it encompasses the target.




```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{pgfviewboxscope}
    {\pgfpoint{5mm}{5mm}}{\pgfpoint{25mm}{15mm}} % Source
    {\pgfpoint{0mm}{0mm}}{\pgfpoint{20mm}{20mm}} % Target
    {meet}
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [radius=8mm] node {Hi};
  \end{pgfviewboxscope} }
```




```
\tikz {
  \draw [red, very thick] (0,0) rectangle (20mm,20mm);
  \begin{pgfviewboxscope}
    {\pgfpoint{5mm}{5mm}}{\pgfpoint{25mm}{15mm}} % Source
    {\pgfpoint{0mm}{0mm}}{\pgfpoint{20mm}{20mm}} % Target
    {slice}
    \draw [blue, very thick] (5mm,5mm) rectangle (25mm,15mm);
    \draw [thick] (1,1) circle [radius=8mm] node {Hi};
  \end{pgfviewboxscope} }
```

\pgfviewboxscope{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  ⟨*environment contents*⟩
\endpgfviewboxscope

The plain TEX version of the environment.

\startpgfviewboxscope{⟨ll_1⟩}{⟨ur_1⟩}{⟨ll_2⟩}{⟨ur_2⟩}{⟨*meet or slice*⟩}
  ⟨*environment contents*⟩
\stoppgfviewboxscope

The ConTEXt version of the environment.

108.4 Nonlinear Transformations

In order to use nonlinear transformations, you first have to load the following PGF module:

```
\usepgfmodule{nonlineartransformations}  % LaTeX and plain TeX when using pure PGF
\usepgfmodule[nonlineartransformations]  % ConTeXt when using pure PGF
```

Loads the necessary functionality for nonlinear transformations.

108.4.1 Introduction

The difference between the coordinate transformations introduced in Section 108.2 above to nonlinear transformations is, of course, that the transformations can be nonlinear. An example of a nonlinear transformation is the transformation underlying polar coordinates: A polar coordinate (r, d) gets transformed to the canvas position $(d \cos r, d \sin r)$, which is clearly not a linear transformation.

Nonlinear transformations work somewhat like the normal linear coordinate transformations in the sense that they apply to coordinate and thereby to the construction of paths, but not to things like text or line width or shadings. (Indeed, it is not possible to apply nonlinear transformations to, say, text.)

This means that there is a fundamental difference between, on the one hand, calling a function like `\pgfpointpolar` or specifying a coordinate as `(45:2)` in Ti*k*Z and, on the other hand, installing the nonlinear transformation "polar coordinates" using the command `\pgftransformnonlinear`: In a coordinate like `(45:2)` the user explicitly says "please evaluate this one coordinate in polar coordinate and then continue in the normal coordinate system with the result". Otherwise nothing changes and a line between two points specified in this way is still a straight line.

Things are quite different when we install a polar *transformation* using `\pgftransformnonlinear`. Now, even a seemingly low-level Cartesian coordinate `\pgfqpoint{1pt}{1pt}` will get transformed. Even more drastically, what is specified as a straight line like

```
\draw (0,1) -- (1,1);
```

can become curved since *everything* gets transformed.

108.4.2 Installing Nonlinear Transformation

\pgftransformnonlinear{⟨*transformation code*⟩}

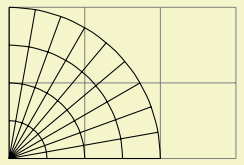
This command adds the ⟨*transformation code*⟩ to the list of non-linear transformations currently in force. Thus, similar to linear coordinate transformations, each additional call to this function adds another transformation to the current TEX scope and the effect ends at the end of the current scope. In practice, however, you typically will not have more than one active nonlinear transformation.

The job of the ⟨*transformation code*⟩ is to map a point p given in the registers `\pgf@x` and `\pgf@y` to a new coordinate $f(p)$, which should be returned in `\pgf@x` and `\pgf@y` as well. As an example, suppose we wish to install polar coordinates as the nonlinear transformation. For this, we need a bit of code:

```
\def\polartransformation{%
  % \pgf@x will contain the radius
  % \pgf@y will contain the distance
  \pgfmathsincos@{\pgf@sys@tonumber\pgf@x}%
  % pgfmathresultx is now the cosine of radius and
  % pgfmathresulty is the sine of radius
  \pgf@x=\pgfmathresultx\pgf@y%
  \pgf@y=\pgfmathresulty\pgf@y%
}
```

(In case you wonder why you cannot just call `\pgfpointpolar` at this point: You can, but this function internally uses `\pgf@x` and `\pgf@y` in complicated ways, so you would first have to safe them so some other registers. Also, the above is faster.)

If we were to call this function again, we would get something funny like "polar-polar coordinates", so let's not do this. Let us instead have a look at the effect this call has: Once a nonlinear transformation is installed, all subsequent path constructions are affected by this transformation. In particular, a normal grid now becomes the typical "polar grid".



```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  % Start nonlinear transformation
  \pgftransformnonlinear{\polartransformation}% see above

  % Draw something with this transformation in force
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);
\end{tikzpicture}
```

108.4.3 Applying Nonlinear Transformations to Points

`\pgfpointtransformednonlinear{`⟨*point*⟩`}`

Works like `\pgfpointtransformed`, but also applies the current nonlinear transformation; that is, it first applies the current linear transformation and then the current nonlinear transformations. Note that, just like `\pgfpointtransformed`, you normally do not call this function directly since it is called internally by the path drawing commands.

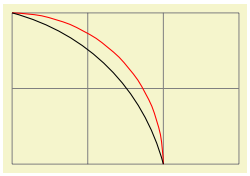
108.4.4 Applying Nonlinear Transformations to Paths

When a nonlinear transformation is installed, the normal path construction commands like `\pgfpathmoveto` get adjusted so that the "honour" the nonlinear transformations currently in force. For `\pgfpathmoveto` this is pretty simple: Instead of just applying the linear transformation matrix to the point to which the path should "jump" next, we also apply the nonlinear transformation. However, for a command like `\pgfpathlineto`, things are much more difficult: A straight line will no longer be a straight line!

In order to make straight lines "bend", the following changes are in force while a nonlinear transformation is installed:

1. Whenever a straight line between two points p and q should be added to the path, either through `\pgfpathlineto` or through `\pgfpathclose`, we replace this straight line by a "degenerated curve" from p to q whose control points are at one third and two third of the distance between p and q on the line between p and q. In this way, while nonlinear transformations are in force, we only need to transform curves.

2. Next, suppose we wish to transform a curve from p to q with supports s and t. For this, we simply apply the nonlinear transformation f to all four points and draw a line with the results. Note that this mapping is actually not quite satisfactory for long lines that are strongly curved:



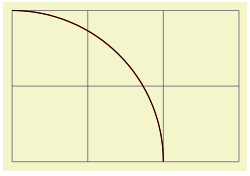
```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgftransformnonlinear{\polartransformation}
    % The curve with the controls computed by pgf: a nice quarter arc
    \draw [red] (0,20mm) -- (90pt,20mm);
  }
  % Here is the curve with controls just transformed:
  \draw (0:20mm) .. controls (30pt:20mm) and (60pt:20mm) .. (90pt:20mm);
\end{tikzpicture}
```

As the example shows, the control points now lie on the arc; but in reality they should point along the tangents at the start and the end. This is exactly when PGF does through the computation described above.

3. To overcome the effect of the control points being "off", it is necessary to split up longer curves into smaller parts, which are drawn individually to increase the accuracy. When such splitting occurs, can be configured using the following command:

   \pgfsettransformnonlinearflatness{⟨*dimension*⟩} (initially 5pt)

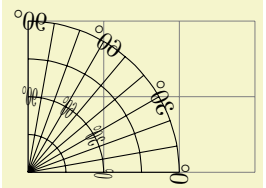
   Whenever in a to-be-drawn curve the L^∞-distance (maximum of the distances in x- and y-directions) between the start of a curve and its first control point or between the first and second control points or between the second control point and the end is more than ⟨*distance*⟩, the curve gets split in the middle (more precisely, at time $t = 0.5$) and we draw the two parts individually (for them, splitting may occur again, if the curve is still too long).



```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \draw[red] (0:20mm) arc [start angle=0, end angle=90, radius=2cm];
  {
    \pgftransformnonlinear{\polartransformation}
    \pgfsettransformnonlinearflatness{2pt} % very precise
    \draw (0,20mm) -- (90pt,20mm);
  }
\end{tikzpicture}
```

108.4.5 Applying Nonlinear Transformations to Text

Earlier, it was pointed that nonlinear transformations do not apply to text. Nevertheless, when you use \pgftext or \pgfnode, PGF will do a sort of "best effort" to render the text in the nonlinear coordinate system: The point where the text should be shown can obviously be computed easily. When then temporarily reset the nonlinear transformation and, instead, setup a linear transformation that matches the nonlinear transformation at the point where the text should be. Then, the text is shown. This means that if the text is longer, it will not "follow" the nonlinear transformation, but near the origin of the text it will look "correct". As an example, let us add some text at the grid point of the above example:



```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \pgftransformnonlinear{\polartransformation}% see above

  % Draw something with this transformation in force
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);

  \foreach \angle in {0,30,60,90}
    \foreach \dist in {1,2}
      {
        \pgftransformshift{\pgfpoint{\angle pt}{\dist cm}}
        \pgftext{\angle$^\circ$}
      }
\end{tikzpicture}
```

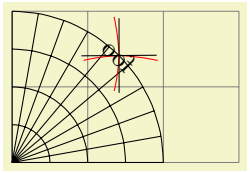
108.4.6 Approximating Nonlinear Transformations Using Linear Transformations

At any given point, the current nonlinear transformation can be approximated using a linear transformation. The following two functions allow you to install such a local approximation:

\pgfapproximatenonlineartransformation

This command will do two things:

1. It clears the nonlinear transformations for the rest of the current TEX scope, so only linear transformations apply.
2. However, before removing the nonlinear transformations, the linear transformation matrix is modified so that it mimics the effect the nonlinear transformation had at the origin. That is, after you call this command, drawing something near the origin will look almost the same as if you had not called it.



```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \pgftransformnonlinear{\polartransformation}% see above
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);

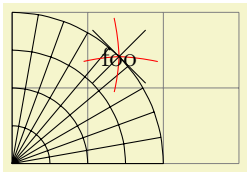
  \begin{scope}[shift={(45pt,20mm)}]
    % Draw something near "origin":
    \draw [red] (-10pt,-10pt) -- (10pt,10pt);
    \draw [red] (10pt,-10pt) -- (-10pt,10pt);

    % Now draw the same, but in the "approximate" coordinate system:
    \pgfapproximatenonlineartransformation
    \draw [] (-10pt,-10pt) -- (10pt,10pt);
    \draw [] (10pt,-10pt) -- (-10pt,10pt);
    \pgftext{foo}
  \end{scope}
\end{tikzpicture}
```

This command is used by \pgftext and \pgfnode to transform text when a nonlinear transformation is in force.

\pgfapproximatenonlineartranslation

This command works like the normal approximation command, but it will only approximate how the origin gets translated, it will not approximate the rotation, skewing, or scaling that is involved. This is useful for drawing text at the right position, but without "mutilating" the text.




```
\usepgfmodule {nonlineartransformations}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  \pgftransformnonlinear{\polartransformation}% see above
  \draw (0pt,0mm) grid [xstep=10pt, ystep=5mm] (90pt, 20mm);

  \begin{scope}[shift={(45pt,20mm)}]
    % Draw something near "origin":
    \draw [red] (-10pt,-10pt) -- (10pt,10pt);
    \draw [red] (10pt,-10pt) -- (-10pt,10pt);

    % Now draw the same, but in the "approximate" coordinate system:
    \pgfapproximatenonlineartranslation
    \draw [] (-10pt,-10pt) -- (10pt,10pt);
    \draw [] (10pt,-10pt) -- (-10pt,10pt);
    \pgftext{foo}
  \end{scope}
\end{tikzpicture}
```

108.4.7 Nonlinear Transformation Libraries

TikZ Library curvilinear

```
\usepgflibrary{curvilinear}  % LaTeX and plain TeX when using pure PGF
\usepgflibrary[curvilinear]  % ConTeXt when using pure PGF
```

```
\usetikzlibrary{curvilinear} % LaTeX and plain TeX when using TikZ
\usetikzlibrary[curvilinear] % ConTeXt when using TikZ
```

This library defines commands for computing nonlinear transformations "along Bézier curves".

Up to now, our running example for a nonlinear transformation was polar transformation. However, is PGF nonlinear transformations are *actually* mainly used for transforming arrow tips; and these need to be transformed "along curves". The `curvilinear` library defines a number of commands that offer the necessary computations for such transformations.

`\pgfsetcurvilinearbeziercurve{`⟨*start*⟩`}{`⟨*first support*⟩`}{`⟨*second support*⟩`}{`⟨*end*⟩`}`

Prior to using any other command from this library, you first call this function to "install" a Bézier curve to which the commands will refer. This curve will be local to the current TeX scope and you can install only one curve at a time.

The main job of this command is to store the passed points internally and to build a lookup table for distance-to-time conversions, see the next command.

```
\pgfsetcurvilinearbeziercurve
  {\pgfpointorigin}
  {\pgfpoint{1cm}{1cm}}
  {\pgfpoint{2cm}{1cm}}
  {\pgfpoint{3cm}{0cm}}
```

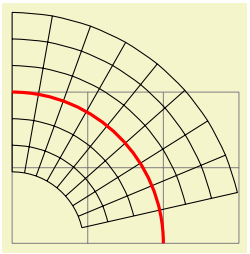
`\pgfcurvilineardistancetotime{`⟨*distance*⟩`}`

This command does a "distance-to-time-conversion": It tries to compute a time t, returned in `\pgf@x`, that corresponds to travelling ⟨*distance*⟩ along the curve that has last been installed using the command `\pgfsetcurvilinearbeziercurve`. The distance-to-time-conversion uses the precomputations done by that command. Note that several compromises had to be made between speed and accuracy:

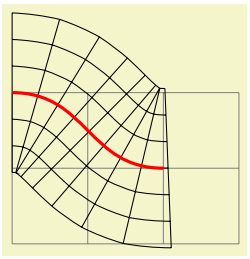
- The conversion will be best near the start of the curve.
- The more "degenerate" the curve, the worse the results.

`\pgfpointcurvilinearbezierorthogonal{`⟨*distance*⟩`}{`⟨*offset*⟩`}`

This command computes the following point: Consider the curve last installed using the command `\pgfsetcurvilinearbeziercurve`. We travel along this curve by ⟨*distance*⟩, arriving at a point p. Then, we turn by 90° and travel by ⟨*offset*⟩ units "always from the curve", arriving at a point q. This point q will now be returned in `\pgf@x` and `\pgf@y`; furthermore, the transformed local coordinate system at point q will also be returned `\pgf@xa` and the other registers, see `\pgftransformnonlinear` for details.



```
\usepgfmodule {nonlineartransformations} \usetikzlibrary {curvilinear}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{11mm}{20mm}}
      {\pgfpoint{20mm}{11mm}}
      {\pgfpoint{20mm}{0mm}}
    \pgftransformnonlinear{\pgfpointcurvilinearbezierorthogonal\pgf@x\pgf@y}%
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
  }
  \draw[red, very thick]
    (0mm,20mm) .. controls (11mm,20mm) and (20mm,11mm) .. (20mm,0mm);
\end{tikzpicture}
```

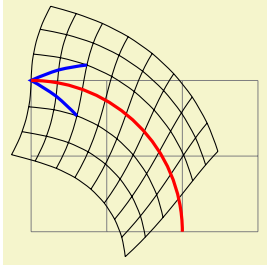


```
\usepgfmodule {nonlineartransformations} \usetikzlibrary {curvilinear}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{10mm}{20mm}}
      {\pgfpoint{10mm}{10mm}}
      {\pgfpoint{20mm}{10mm}}
    \pgftransformnonlinear{\pgfpointcurvilinearbezierorthogonal\pgf@x\pgf@y}%
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
  }
  \draw[red, very thick]
    (0mm,20mm) .. controls (10mm,20mm) and (10mm,10mm) .. (20mm,10mm);
\end{tikzpicture}
```

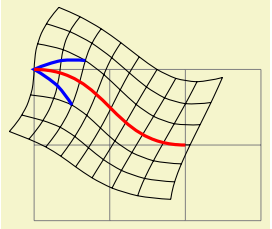
\pgfpointcurvilinearbezierpolar{⟨x⟩}{⟨y⟩}

This command is similar to the previous version, but the transformation is different: The idea is that a line form $(0,0)$ to $(x,0)$ gets transformed to the curve from the start of the curve to a point at distance x along the curve. This is identical to what the "orthogonal" transformation above also does. The difference is that a line from $(0,0)$ to $(0,y)$ gets still transformed to an initial segment of the curve of a length of y, but now rotated by 90°. In general, the point $p=(x,y)$ gets transferred to a point that at distance $|p|=\sqrt{x^2+y^2}$ along the curve, but rotated by the angle of p relative to the x-axis.

All of these computations mainly have the following effect: Two straight lines from the start of the curve as in a `Straight Barb` arrow tip get transformed to an initial segment of the curve whose length is the length of the two lines, but this segment gets rotated by the angle of the two lines.



```
\usepgfmodule {nonlineartransformations} \usetikzlibrary {curvilinear}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{11mm}{20mm}}
      {\pgfpoint{20mm}{11mm}}
      {\pgfpoint{20mm}{0mm}}
    \pgftransformnonlinear{\pgfpointcurvilinearbezierpolar\pgf@x\pgf@y}%
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
    % Add a "barb":
    \draw [blue, very thick] (20pt,10pt) -- (0,0) -- (20pt,-10pt);
  }
  \draw[red, very thick]
    (0mm,20mm) .. controls (11mm,20mm) and (20mm,11mm) .. (20mm,0mm);
\end{tikzpicture}
```



```
\usepgfmodule {nonlineartransformations} \usetikzlibrary {curvilinear}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (3,2);
  {
    \pgfsetcurvilinearbeziercurve
      {\pgfpoint{0mm}{20mm}}
      {\pgfpoint{10mm}{20mm}}
      {\pgfpoint{10mm}{10mm}}
      {\pgfpoint{20mm}{10mm}}
    \pgftransformnonlinear{\pgfpointcurvilinearbezierpolar\pgf@x\pgf@y}%
    \draw (0,-30pt) grid [step=10pt] (80pt,30pt);
    % Add a "barb":
    \draw [blue, very thick] (20pt,10pt) -- (0,0) -- (20pt,-10pt);
  }
  \draw[red, very thick]
    (0mm,20mm) .. controls (10mm,20mm) and (10mm,10mm) .. (20mm,10mm);
\end{tikzpicture}
```

109 Patterns

109.1 Overview

There are many ways of filling a path. First, you can fill it using a solid color and this is also the fastest method. Second, you can also fill it using a shading, which means that the color changes smoothly between two (or more) different colors. Third, you can fill it using a tiling pattern and it is explained in the following how this is done.

A tiling pattern can be imagined as a rectangular tile (hence the name) on which a small picture is painted. There is not a single tile, but (conceptually) an infinite number of tiles, all showing the same picture, and these tiles are arranged horizontally and vertically to fill the plane. When you use a tiling pattern to fill a path, what happens is that the path clips out a "window" through which we see part of this infinite plane.

Patterns come in two versions: *inherently colored patterns* and *form-only patterns*. (These are often called "color patterns" and "uncolored patterns", but these names are misleading since uncolored patterns do have a color and the color changes. As I said, the name is misleading…) An inherently colored pattern is just a colored tile like, say, a red star with a black outline. A form-only pattern can be imagined as a tile that is a kind of rubber stamp. When this pattern is used, the stamp is used to print copies of the stamp picture onto the plane, but we can use a different stamp color each time we use a form-only pattern.

PGF provides a special support for patterns. You can declare a pattern and then use it very much like a fill color. PGF directly maps patterns to the pattern facilities of the underlying graphic languages (PostScript, PDF, and SVG). This means that filling a path using a pattern will be nearly as fast as if you used a uniform color.

There are a number of pitfalls and restrictions when using patterns. First, once a pattern has been declared, you cannot change it anymore. In particular, it is not possible to enlarge it or change the line width. Such flexibility would require that the repeating of the pattern were not done by the graphic language, but on the PGF level. This would make patterns orders of magnitude slower to produce and to render. However, PGF does provide a more-or-less successful emulation of "mutable" patterns, although internally, a new (fixed) instance of a pattern is declared when the parameters of a pattern change.

Second, the phase of patterns is not well-defined, that is, it is not clear where the origin of the "first" tile is. To be more precise, PostScript and PDF on the one hand and SVG on the other hand define the origin differently. PostScript and PDF define a fixed origin that is independent of where the path lies. This has the highly desirable effect that if you use the same pattern to fill multiple paths, the outcome is the same as if you had filled a single path consisting of the union of all these paths. By comparison, SVG uses the upper-left (?) corner of the path to be filled as the origin. However, the SVG specification is a bit vague on this question.

109.2 Declaring a Pattern

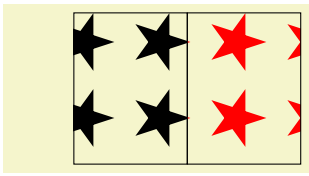
Before a pattern can be used, it must be declared. The following command is used for this:

`\pgfdeclarepatternformonly[`⟨*variables*⟩`]{`⟨*name*⟩`}{`⟨*bottom left*⟩`}{`⟨*top right*⟩`}{`⟨*tile size*⟩`}{`⟨*code*⟩`}`

This command declares a new form-only pattern. The ⟨*name*⟩ is a name for later reference. The two parameters ⟨*lower left*⟩ and ⟨*upper right*⟩ must describe a bounding box that is large enough to encompass the complete tile.

The size of a tile is given by ⟨*tile size*⟩, that is, a tile is a rectangle whose lower left corner is the origin and whose upper right corner is given by ⟨*tile size*⟩. This might make you wonder why the second and third parameters are needed. First, the bounding box might be smaller than the tile size if the tile is larger than the picture on the tile. Second, the bounding box might be bigger, in which case the picture will "bleed" over the tile.

The ⟨*code*⟩ should be PGF code that can be protocolled. It should not contain any color code.



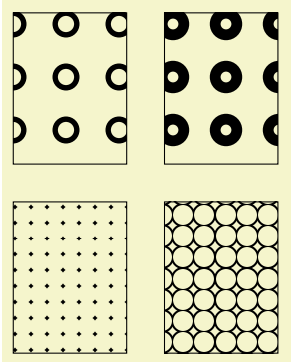
```
\usetikzlibrary {patterns}
\pgfdeclarepatternformonly{stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
{
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{fill}
}
\begin{tikzpicture}
  \filldraw[pattern=stars] (0,0)   rectangle (1.5,2);
  \filldraw[pattern=stars,pattern color=red]
                           (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

The optional argument ⟨*variables*⟩ consists of a comma separated list of macros, registers or keys, representing the parameters of the pattern that may vary. If a variable is a key, then the full path name must be used (specifically, it must start with `/`). As an example, a list might look like the following: `\mymacro,\mydimen,/pgf/my key`. Note that macros and keys should be "simple". They should only store values in themselves.

The effect of ⟨*variables*⟩, is the following: Normally, when this argument is empty, once a pattern has been declared, it becomes "frozen". This means that it is not possible to enlarge the pattern or change the line width later on. By specifying ⟨*variables*⟩, no pattern is actually created. Instead, the arguments are stored away (so the macros, registers or keys do not have to be defined in advance).

When the fill pattern is set, PGF checks if the pattern has already been created with the ⟨*variables*⟩ set to their current values (PGF is usually "smart enough" to distinguish between macros, registers and keys). If so, this already-declared-pattern is used as the fill pattern. If not, a new instance of the pattern (which will have a unique internal name) is declared using the current values of ⟨*variables*⟩. These values are then saved and the fill pattern set accordingly.

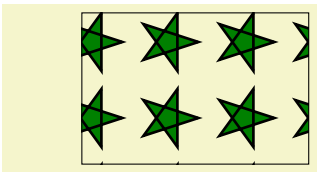
The following shows an example of a pattern which varies according to the values of the macro `\size`, the key `/tikz/radius`, and the TEX dimension `\thickness`.



```
\usetikzlibrary {patterns}
\pgfdeclarepatternformonly[/tikz/radius,\thickness,\size]{rings}
{\pgfpoint{-0.5*\size}{-0.5*\size}}
{\pgfpoint{0.5*\size}{0.5*\size}}
{\pgfpoint{\size}{\size}}
{
  \pgfsetlinewidth{\thickness}
  \pgfpathcircle\pgfpointorigin{\pgfkeysvalueof{/tikz/radius}}
  \pgfusepath{stroke}
}
\newdimen\thickness
\tikzset{
  radius/.initial=4pt,
  size/.store in=\size, size=20pt,
  thickness/.code={\thickness=#1},
  thickness=0.75pt
}
\begin{tikzpicture}[rings/.style={pattern=rings}]
  \filldraw [rings, radius=2pt, size=6pt]      (0,0)   rectangle +(1.5,2);
  \filldraw [rings, radius=2pt, size=8pt]      (2,0)   rectangle +(1.5,2);
  \filldraw [rings, radius=6pt, thickness=2pt] (0,2.5) rectangle +(1.5,2);
  \filldraw [rings, radius=8pt, thickness=4pt] (2,2.5) rectangle +(1.5,2);
\end{tikzpicture}
```

`\pgfdeclarepatterninherentlycolored[`⟨*variables*⟩`]` `{`⟨*name*⟩`}` `{`⟨*lower left*⟩`}` `{`⟨*upper right*⟩`}` `{`⟨*tile size*⟩`}` `{`⟨*code*⟩`}`

This command works like `\pgfdeclarepatternuncolored`, only the pattern will have an inherent color. To set the color, you should use PGF's color commands, not the `\color` command, since this fill is not protocolled.



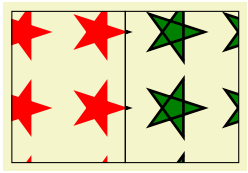
```
\usetikzlibrary {patterns}
\pgfdeclarepatterninherentlycolored{green stars}
{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
{\pgfpoint{1cm}{1cm}}
{
  \pgfsetfillcolor{green!50!black}
  \pgftransformshift{\pgfpoint{.5cm}{.5cm}}
  \pgfpathmoveto{\pgfpointpolar{0}{4mm}}
  \pgfpathlineto{\pgfpointpolar{144}{4mm}}
  \pgfpathlineto{\pgfpointpolar{288}{4mm}}
  \pgfpathlineto{\pgfpointpolar{72}{4mm}}
  \pgfpathlineto{\pgfpointpolar{216}{4mm}}
  \pgfpathclose%
  \pgfusepath{stroke,fill}
}
\begin{tikzpicture}
  \filldraw[pattern=green stars] (0,0) rectangle (3,2);
\end{tikzpicture}
```

109.3 Setting a Pattern

Once a pattern has been declared, it can be used.

\pgfsetfillpattern{⟨*name*⟩}{⟨*color*⟩}

This command specifies that paths that are filled should be filled with the "color" by the pattern ⟨*name*⟩. For an inherently colored pattern, the ⟨*color*⟩ parameter is ignored. For form-only patterns, the ⟨*color*⟩ parameter specifies the color to be used for the pattern.



```
\usetikzlibrary {patterns} \pgfdeclarepatternformonly {stars} {\pgfpointorigin }{\pgfpoint
{1cm}{1cm}} {\pgfpoint {1cm}{1cm}} { \pgftransformshift {\pgfpoint {.5cm}{.5cm}}
\pgfpathmoveto {\pgfpointpolar {0}{4mm}} \pgfpathlineto {\pgfpointpolar {144}{4mm}}
\pgfpathlineto {\pgfpointpolar {288}{4mm}} \pgfpathlineto {\pgfpointpolar {72}{4mm}}
\pgfpathlineto {\pgfpointpolar {216}{4mm}} \pgfpathclose \pgfusepath {fill} }
\pgfdeclarepatterninherentlycolored {green stars} {\pgfpointorigin }{\pgfpoint {1cm}{1cm}}
{\pgfpoint {1cm}{1cm}} { \pgfsetfillcolor {green!50!black} \pgftransformshift {\pgfpoint
{.5cm}{.5cm}} \pgfpathmoveto {\pgfpointpolar {0}{4mm}} \pgfpathlineto {\pgfpointpolar
{144}{4mm}} \pgfpathlineto {\pgfpointpolar {288}{4mm}} \pgfpathlineto {\pgfpointpolar
{72}{4mm}} \pgfpathlineto {\pgfpointpolar {216}{4mm}} \pgfpathclose \pgfusepath
{stroke,fill} }
\begin{tikzpicture}
  \pgfsetfillpattern{stars}{red}
  \filldraw (0,0) rectangle (1.5,2);

  \pgfsetfillpattern{green stars}{red}
  \filldraw (1.5,0) rectangle (3,2);
\end{tikzpicture}
```

110 Declaring and Using Images

This section describes the commands for creating images.

110.1 Overview

To be quite frank, LaTeX's `\includegraphics` is designed better than PGF's image mechanism. For this reason, *I recommend that you use the standard image inclusion mechanism of your format.* Thus, LaTeX users are encouraged to use `\includegraphics` to include images.

However, there are reasons why you might need to use the image inclusion facilities of PGF:

- There is no standard image inclusion mechanism in your format. For example, plain TeX does not have one, so PGF's inclusion mechanism is "better than nothing".

  However, this applies only to the `pdftex` backend. For all other backends, PGF currently maps its commands back to the `graphicx` package. Thus, in plain TeX, this does not really help. It might be a good idea to fix this in the future such that PGF becomes independent of LaTeX, thereby providing a uniform image abstraction for all formats.

- You wish to use masking. This is a feature that is only supported by PGF, though I hope that someone will implement this also for the graphics package in LaTeX in the future.

Whatever your choice, you can still use the usual image inclusion facilities of the `graphics` package.

The general approach taken by PGF to including an image is the following: First, `\pgfdeclareimage` declares the image. This must be done prior to the first use of the image. Once you have declared an image, you can insert it into the text using `\pgfuseimage`. The advantage of this two-phase approach is that, at least for PDF, the image data will only be included once in the file. This can drastically reduce the file size if you use an image repeatedly, for example in an overlay. However, there is also a command called `\pgfimage` that declares and then immediately uses the image.

To speedup the compilation, you may wish to use the following class option:

`\usepackage[draft]{pgf}`

In draft mode boxes showing the image name replace the images. It is checked whether the image files exist, but they are not read. If either height or width is not given, 1cm is used instead.

110.2 Declaring an Image

`\pgfdeclareimage[`⟨*options*⟩`]{`⟨*image name*⟩`}{`⟨*filename*⟩`}`

Declares an image, but does not paint anything. To draw the image, use `\pgfuseimage{`⟨*image name*⟩`}`. The ⟨*filename*⟩ may not have an extension. For PDF, the extensions `.pdf`, `.jpg`, and `.png` will automatically tried. For PostScript, the extensions `.eps`, `.epsi`, and `.ps` will be tried.

The following options are possible:

- `height=`⟨*dimension*⟩ sets the height of the image. If the width is not specified simultaneously, the aspect ratio of the image is kept.
- `width=`⟨*dimension*⟩ sets the width of the image. If the height is not specified simultaneously, the aspect ratio of the image is kept.
- `page=`⟨*page number*⟩ selects a given page number from a multipage document. Specifying this option will have the following effect: first, PGF tries to find a file named

  ⟨*filename*⟩`.page`⟨*page number*⟩`.`⟨*extension*⟩

  If such a file is found, it will be used instead of the originally specified filename. If not, PGF inserts the image stored in ⟨*filename*⟩`.`⟨*extension*⟩ and if a recent version of `pdflatex` is used, only the selected page is inserted. For older versions of `pdflatex` and for `dvips` the complete document is inserted and a warning is printed.
- `interpolate=`⟨*true or false*⟩ selects whether the image should be "smoothed" when zoomed. False by default.
- `mask=`⟨*mask name*⟩ selects a transparency mask. The mask must previously be declared using `\pgfdeclaremask` (see below). This option only has an effect for `pdf`. Not all viewers support masking.

```
\pgfdeclareimage[interpolate=true,height=1cm]{image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
```

\pgfaliasimage{⟨*new image name*⟩}{⟨*existing image name*⟩}

The {⟨*existing image name*⟩} is "cloned" and the {⟨*new image name*⟩} can now be used whenever the original image is used. This command is useful for creating aliases for alternate extensions and for accessing the last image inserted using \pgfimage.

Example: \pgfaliasimage{image.!30!white}{image.!25!white}

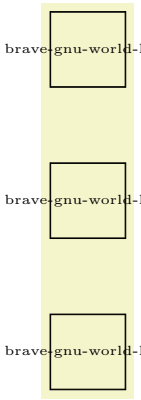
110.3 Using an Image

\pgfuseimage{⟨*image name*⟩}

Inserts a previously declared image into the *normal text*. If you wish to use it in a {pgfpicture} environment, you must put a \pgftext around it.

If the macro \pgfalternateextension expands to some nonempty ⟨*alternate extension*⟩, PGF will first try to use the image named ⟨*image name*⟩.⟨*alternate extension*⟩. If this image is not defined, PGF will next check whether ⟨*alternate extension*⟩ contains a ! character. If so, everything up to this exclamation mark and including it is deleted from ⟨*alternate extension*⟩ and the PGF again tries to use the image ⟨*image name*⟩.⟨*alternate extension*⟩. This is repeated until ⟨*alternate extension*⟩ no longer contains a !. Then the original image is used.

The xxcolor package sets the alternate extension to the current color mixin.



```
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
  {image1}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,width=1cm]{image2}{brave-gnu-world-logo}
\pgfdeclareimage[interpolate=true,height=1cm]{image3}{brave-gnu-world-logo}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
```

The following example demonstrates the effect of using \pgfuseimage inside a colormixin environment.



```
\usepackage {xxcolor}
\pgfdeclareimage[interpolate=true,width=1cm,height=1cm]
  {image1.!25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,width=1cm]
  {image2.25!white}{brave-gnu-world-logo.25}
\pgfdeclareimage[interpolate=true,height=1cm]
  {image3.white}{brave-gnu-world-logo.25}
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]{\pgfuseimage{image1}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]{\pgfuseimage{image2}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]{\pgfuseimage{image3}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

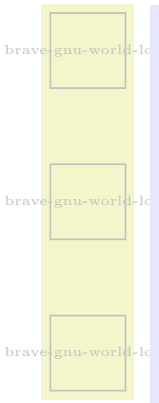
\pgfalternateextension

You should redefine this command to install a different alternate extension.

Example: `\def\pgfalternateextension{!25!white}`

`\pgfimage[`⟨*options*⟩`]{`⟨*filename*⟩`}`

Declares the image under the name `pgflastimage` and immediately uses it. You can "save" the image for later usage by invoking `\pgfaliasimage` on `pgflastimage`.



```
\usepackage {xxcolor}
\begin{colormixin}{25!white}
\begin{pgfpicture}
  \pgftext[at=\pgfpoint{1cm}{5cm},left,base]
    {\pgfimage[interpolate=true,width=1cm,height=1cm]{brave-gnu-world-logo}}
  \pgftext[at=\pgfpoint{1cm}{3cm},left,base]
    {\pgfimage[interpolate=true,width=1cm]{brave-gnu-world-logo}}
  \pgftext[at=\pgfpoint{1cm}{1cm},left,base]
    {\pgfimage[interpolate=true,height=1cm]{brave-gnu-world-logo}}

  \pgfpathrectangle{\pgfpoint{1cm}{5cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{3cm}}{\pgfpoint{1cm}{1cm}}
  \pgfpathrectangle{\pgfpoint{1cm}{1cm}}{\pgfpoint{1cm}{1cm}}
  \pgfusepath{stroke}
\end{pgfpicture}
\end{colormixin}
```

110.4 Masking an Image

`\pgfdeclaremask[`⟨*options*⟩`]{`⟨*mask name*⟩`}{`⟨*filename*⟩`}`

Declares a transparency mask named ⟨*mask name*⟩ (called a *soft mask* in the PDF specification). This mask is read from the file ⟨*filename*⟩. This file should contain a grayscale image that is as large as the actual image. A white pixel in the mask will correspond to "transparent", a black pixel to "solid", and gray values correspond to intermediate values. The mask must have a single "color channel". This means that the mask must be a "real" grayscale image, not an RGB-image in which all RGB-triples happen to have the same components.

You can only mask images that are in a "pixel format". For drivers with PDF output, these are `.jpg` and `.png` image files; you cannot mask `.pdf` images in this way. Pixel images for the `dvips+ps2pdf` workflow must be provided as `.eps` or `.ps` files. Also, again, the mask file and the image file must have the same size.

The following options may be given:

- `matte={`⟨*color components*⟩`}` sets the so-called *matte* of the actual image (strangely, this has to be specified together with the mask, not with the image itself). The matte is the color that has been used to preblend the image. For example, if the image has been preblended with a red background, then ⟨*color components*⟩ should be set to `{1 0 0}`. The default is `{1 1 1}`, which is white in the rgb model.

  The matte is specified in terms of the parent's image color space. Thus, if the parent is a grayscale image, the matte has to be set to `{1}`.

Example:



```
%% Draw a large colorful background
\pgfdeclarehorizontalshading{colorful}{5cm}{color(0cm)=(red);
color(2cm)=(green); color(4cm)=(blue); color(6cm)=(red);
color(8cm)=(green); color(10cm)=(blue); color(12cm)=(red);
color(14cm)=(green)}
\hbox{\pgfuseshading{colorful}\hskip-14cm\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo}\hskip1cm
\pgfimage[height=4cm]{brave-gnu-world-logo-mask}\hskip1cm
\pgfdeclaremask{mymask}{brave-gnu-world-logo-mask}
\pgfimage[mask=mymask,height=4cm,interpolate=true]{brave-gnu-world-logo}}
```

111 Externalizing Graphics

111.1 Overview

There are two fundamentally different ways of inserting graphics into a TEX-document. First, you can create a graphic using some external program like `xfig` or `InDesign` and then include this graphic in your text. This is done using commands like `\includegraphics` or `\pgfimage`. In this case, the graphic file contains all the low-level graphic commands that describe the picture. When such a file is included, all TEX has to worry about is the size of the picture; the internals of the picture are unknown to TEX and it does not care about them.

The second method of creating graphics is to use a special package that transforms TEX-commands like `\draw` or `\psline` into appropriate low-level graphic commands. In this case, TEX has to do all the hard work of "typesetting" the picture and if a picture has a complicated internal structure this may take a lot of time.

While PGF was created to facilitate the second method of creating pictures, there are two main reasons why you may need to employ the first method of image-inclusion, nevertheless:

1. Typesetting a picture using TEX can be a very time-consuming process. If TEX needs a minute to typeset a picture, you do not want to wait this minute when you reTEX your document after having changed a single comma.

2. Some users, especially journal editors, may not be able to process files that contain PGF commands – for the simple reason that the systems of many publishing houses do not have PGF installed.

In both cases, the solution is to "extract" or "externalize" pictures that would normally be typeset every time a document is TEXed. Once the pictures have been extracted into separate graphics files, these graphic files can be reinserted into the text using the first method.

Extracting a graphic from a file is not as easy as it may sound at first since TEX cannot write parts of its output into different files and a bit of trickery is needed. The following macros simplify the workflow:

1. You have to tell PGF which files will be used for which pictures. To do so, you enclose each picture that you wish to be "externalized" in a pair of `\beginpgfgraphicnamed` and `\endpgfgraphicnamed` macros.

2. The next step is to generate the extracted graphics. For this you run TEX with the `\jobname` set to the graphic file's name. This will cause `\pgfname` to behave in a very special way: All of your document will simply be thrown away, *except* for the single graphic having the same name as the current jobname.

3. After you have run TEX once for each graphic that your wish to externalize, you can rerun TEX on your document normally. This will have the following effect: Each time a `\beginpgfgraphicnamed` is encountered, PGF checks whether a graphic file of the given name exists (if you did step 2, it will). If this graphic file exists, it will be input and the text till the corresponding `\endpgfgraphicnamed` will be ignored.

In the rest of this section, the above workflow is explained in more detail.

111.2 Workflow Step 1: Naming Graphics

In order to put each graphic in an external file, you first need to tell PGF the names of these files.

`\beginpgfgraphicnamed{`⟨*file name prefix*⟩`}`

This command indicates that everything up to the next call of `\endpgfgraphicnamed` is part of a graphic that should be placed in a file named ⟨*file name prefix*⟩`.`⟨*suffix*⟩, where the ⟨*suffix*⟩ depends on your backend driver. Typically, ⟨*suffix*⟩ will be `dvi` or `pdf`.

Here is a typical example of how this command is used:

```
% In file main.tex:
...
As we see in Figure~\ref{fig1}, the world is flat.
\begin{figure}
  \beginpgfgraphicnamed{graphic-of-flat-world}
  \begin{tikzpicture}
    \fill (0,0) circle [radius=1cm];
  \end{tikzpicture}
  \endpgfgraphicnamed
  \caption{The flat world.}
  \label{fig1}
\end{figure}
```

Each graphic to be externalized should have a unique name. Note that this name will be used as the name of a file in the file system, so it should not contain any funny characters.

This command can have three different effects:

1. The easiest situation arises if there does not yet exist a graphic file called ⟨*file name prefix*⟩.⟨*suffix*⟩, where the ⟨*suffix*⟩ is one of the suffixes understood by your current backend driver (so `pdf` or `jpg` if you use `pdftex`, `eps` if you use `dvips`, and so on). In this case, both this command and the `\endpgfgraphicnamed` command simply have no effect.
2. A more complex situation arises when a graphic file named ⟨*file name prefix*⟩.⟨*suffix*⟩ *does* exist. In this case, this graphic file is included using the `\includegraphics` command[16]. Furthermore, the text between `\beginpgfgraphicnamed` and `\endpgfgraphicnamed` is ignored.

   When the text is "ignored", what actually happens is that all text up to the next occurrence of `\endpgfgraphicnamed` is thrown away without any macro expansion. This means, in particular, that (a) you cannot put `\endpgfgraphicnamed` inside a macro and (b) the macros used in the graphics need not be defined at all when the graphic file is included.
3. The most complex behavior arises when current the `\jobname` equals the ⟨*file name prefix*⟩ and, furthermore, the *real job name* has been declared. The behavior for this case is explained later.

Note that the `\beginpgfgraphicnamed` does not really have any effect until you have generated the graphic files named. Till then, this command is simply ignored. Also, if you delete the graphics file later on, the graphics are typeset normally once more.

`\endpgfgraphicnamed`

This command just marks the end of the graphic that should be externalized.

111.3 Workflow Step 2: Generating the External Graphics

We have now indicated all the graphics for which we would like graphic files to be generated. In order to generate the files, you now need to modify the `\jobname` appropriately. This is done in two steps:

1. You use the following command to tell PGF the real name of your `.tex` file:

   `\pgfrealjobname{`⟨*name*⟩`}`

   Tells PGF the real name of your job. For instance, if you have a file called `survey.tex` that contains two graphics that you wish to be called `survey-graphic1` and `survey-graphic2`, then you should write the following.

   ```
   % This is file survey.tex
   \documentclass{article}
   ...
   \usepackage{tikz}
   \pgfrealjobname{survey}
   ```

2. You run TEX with the `\jobname` set to the name of the graphic for which you need an external graphic to be generated. To set the `\jobname`, you use the `--jobname=` option of TEX:

   ```
   bash> latex --jobname=survey-graphic1 survey.tex
   ```

[16] Actually, the command key `/pgf/images/include external` is invoked which calls an appropriate `\includegraphics` command.

The following things will now happen:

1. `\pgfrealjobname` notices that the `\jobname` is not the "real" jobname and, thus, must be the name of a graphic that is to be put in an external file.
2. At the beginning of the document, PGF changes the definition of TEX's internal `\shipout` macro. The new shipout macro simply throws away the output. This means that the document is typeset normally, but no output is produced.
3. When the `\beginpgfgraphicnamed{`⟨*name*⟩`}` command is encountered where the ⟨*name*⟩ is the same as the current `\jobname`, then a TEX-box is started and ⟨*everything*⟩ up to the following `\endpgfgraphicnamed` command is stored inside this box.

   Note that, typically, ⟨*everything*⟩ will contain just a single `{tikzpicture}` or `{pgfpicture}` environment. However, this need not be the case, you can use, say, a `{pspicture}` environment as ⟨*everything*⟩ or even just some normal TEX-text.
4. At the `\endpgfgraphicnamed`, the box *is* shipped out using the original `\shipout` command. Thus, unlike everything else, the contents of the graphic is made part of the output.
5. When the box containing the graphic is shipped out, the paper size is modified such that it is exactly equal to the height and width of the box.

The net effect of everything described above is that the two commands

```
bash> latex --jobname=survey-graphic1 survey.tex
bash> dvips survey-graphic1
```

produce a file called `survey-graphic1.ps` that consists of a single page that contains exactly the graphic produced by the code between `\beginpgfgraphicnamed{survey-graphic1}` and `\endpgfgraphicnamed`. Furthermore, the size of this single page is exactly the size of the graphic.

If you use pdfTEX, producing the graphic is even simpler:

```
bash> pdflatex --jobname=survey-graphic1 survey.tex
```

produces the single-page `pdf`-file `survey-graphic1.pdf`.

111.4 Workflow Step 3: Including the External Graphics

Once you have produced all the pictures in the text, including them into the main document is easy: Simply run TEX again without any modification of the `\jobname`. In this case the `\pgfrealjobname` command will notice that the main file is, indeed, the main file. The main file will then be typeset normally and the `\beginpgfgraphicnamed` commands also behave normally, which means that they will try to include the generated graphic files – which is exactly what you want.

Suppose that you wish to send your survey to a journal that does not have PGF installed. In this case, you now have all the necessary external graphics, but you still need PGF to automatically include them instead of the executing the picture code! One way to solve this problem is to simply delete all of the PGF or Ti*k*Z code from your `survey.tex` and instead insert appropriate `\includegraphics` commands "by hand". However, there is a better way: You input the file `pgfexternal.tex`.

File pgfexternal.tex

This file defines the command `\beginpgfgraphicnamed` and causes it to have the following effect: It includes the graphic file given as a parameter to it and then gobbles everything up to `\endpgfgraphicnamed`.

Since `\beginpgfgraphicnamed` does not do macro expansion as it searches for `\endpgfgraphicnamed`, it is not necessary to actually include the packages necessary for *creating* the graphics. So the idea is that you comment out things like `\usepackage{tikz}` and instead say `\input{pgfexternal.tex}`.

Indeed, the contents of this file is simply the following line:

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
```

Instead of `\input{pgfexternal.tex}` you could also include this line in your main file.

As a final remark, note that the `baseline` option does not work directly with pictures written to an external graphic file. The simple reason is that there is no way to store this baseline information in an external graphic file. To allow the `baseline` option (or any TEX construction with non-zero depth), the baseline information is stored into a separate file. This file is named `{`⟨*image file*⟩`}.dpth` and contains something like `5pt`.

So, if you need baseline information, you will have to keep the external graphic file together with its `.dpth` file. Furthermore, the short command in `\input{pgfexternal.tex}` is no longer enough because it ignores any baseline information. You will need to use `\input{pgfexternalwithdepth.tex}` instead (it is shown below). It is slightly longer, but it can be used in the same way as `pgfexternal.tex`.

`/pgf/images/include external` (initially `\pgfimage{#1}`)

This key constitutes the public interface to exchange the `\includegraphics` command used for the image inclusion.

Redefining this key allows to provide bounding box or viewport options:

```
\pgfkeys{/pgf/images/include external/.code={\includegraphics[viewport=0 0 211.28 175.686]{#1}}}
```

Do not forget the `.code` here which redefines the command.

One application could be image externalization and bounding box restrictions: As far as I know, a `.pdf` graphics with restricted bounding box is always cropped (which is not always desired). One solution could be to use `latex` and `dvips` which doesn't have this restriction. Another is to manually provide the `viewport` option as shown above.

A possible value for `viewport` can be found in the `.pdf` image, search for `/MediaBox = [ ... ]`.

111.5 A Complete Example

Let us now have a look at a simple, but complete example. We start out with a normal file called `survey.tex` that has the following contents:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\usepackage{tikz}

\begin{document}
In the following figure, we see a circle:
\begin{tikzpicture}
  \fill (0,0) circle [radius=10pt];
\end{tikzpicture}

By comparison, in this figure we see a rectangle:
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\end{document}
```

Now our editor tells us that the publisher will need all figures to be provided in separate PostScript or `.pdf`-files. For this, we enclose all figures in `...graphicnamed`-pairs and we add a call to the `\pgfrealjobname` macro:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\usepackage{tikz}
\pgfrealjobname{survey}

\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle [radius=10pt];
\end{tikzpicture}
\endpgfgraphicnamed

By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

After these changes, typesetting the file will still yield the same output as it did before – after all, we have not yet created any external graphics.

To create the external graphics, we run `pdflatex` twice, once for each graphic:

```
bash> pdflatex --jobname=survey-f1 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
) [1] (./survey-f1.aux) )
Output written on survey-f1.pdf (1 page, 1016 bytes).
Transcript written on survey-f1.log.
```

```
bash> pdflatex --jobname=survey-f2 survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
...
(./survey-f2.aux) )
Output written on survey-f2.pdf (1 page, 1002 bytes).
Transcript written on survey-f2.log.
```

We can now send the two generated graphics (`survey-f1.pdf` and `survey-f2.pdf`) to the editor. However, the publisher cannot use our `survey.tex` file, yet. The reason is that it contains the command `\usepackage{tikz}` and they do not have PGF installed.

Thus, we modify the main file `survey.tex` as follows:

```
% This is the file survey.tex
\documentclass{article}

\usepackage{graphics}
\input{pgfexternal.tex}
% \usepackage{tikz}
% \pgfrealjobname{survey}

\begin{document}
In the following figure, we see a circle:
\beginpgfgraphicnamed{survey-f1}
\begin{tikzpicture}
  \fill (0,0) circle [radius=10pt];
\end{tikzpicture}
\endpgfgraphicnamed

By comparison, in this figure we see a rectangle:
\beginpgfgraphicnamed{survey-f2}
\begin{tikzpicture}
  \fill (0,0) rectangle (10pt,10pt);
\end{tikzpicture}
\endpgfgraphicnamed
\end{document}
```

If we now run pdfLaTeX, then, indeed, PGF is no longer needed:

```
bash> pdflatex survey.tex
This is pdfTeX, Version 3.141592-1.40.3 (Web2C 7.5.6)
entering extended mode
(./survey.tex
LaTeX2e <2005/12/01>
Babel <v3.8h> and hyphenation patterns for english, ..., loaded.
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/article.cls
Document Class: article 2005/09/16 v1.4f Standard LaTeX document class
(/usr/local/gwTeX/texmf.texlive/tex/latex/base/size10.clo))
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/graphics.sty
(/usr/local/gwTeX/texmf.texlive/tex/latex/graphics/trig.sty)
(/usr/local/gwTeX/texmf.texlive/tex/latex/config/graphics.cfg)
(/usr/local/gwTeX/texmf.texlive/tex/latex/pdftex-def/pdftex.def))
(/Users/tantau/Library/texmf/tex/generic/pgf/generic/pgf/utilities/pgfexternal.
tex) (./survey.aux)
(/usr/local/gwTeX/texmf.texlive/tex/context/base/supp-pdf.tex
[Loading MPS to PDF converter (version 2006.09.02).]
) <survey-f1.pdf, id=1, 23.33318pt x 19.99973pt> <use survey-f1.pdf>
<survey-f2.pdf, id=2, 13.33382pt x 10.00037pt> <use survey-f2.pdf> [1{/Users/ta
ntau/Library/texmf/fonts/map/pdftex/updmap/pdftex.map} <./survey-f1.pdf> <./sur
vey-f2.pdf>] (./survey.aux) )</usr/local/gwTeX/texmf.texlive/fonts/type1/bluesk
y/cm/cmr10.pfb>
Output written on survey.pdf (1 page, 10006 bytes).
Transcript written on survey.log.
```

To our editor, we send the following files:

- The last `survey.tex` shown above.
- The graphic file `survey-f1.pdf`.
- The graphic file `survey-f2.pdf`.
- The file `pgfexternal.tex`, whose contents is simply

  ```
  \long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{\includegraphics{#1}}
  ```

  (Alternatively, we can also directly add this line to our `survey.tex` file).

In case we have used the `baseline` option, we also need to include any `.dpth` files and we need to use the file `pgfexternalwithdepth.tex` instead of `pgfexternal.tex`. This file also checks for the existence of `.dpth` files containing baseline information, its contents is

```
\long\def\beginpgfgraphicnamed#1#2\endpgfgraphicnamed{%
    \begingroup
    \setbox1=\hbox{\includegraphics{#1}}%
    \openin1=#1.dpth
    \ifeof1 \box1
    \else
        \read1 to\pgfincludeexternalgraphicsdp\closein1
        \dimen0=\pgfincludeexternalgraphicsdp\relax
        \hbox{\lower\dimen0 \box1 }%
    \fi
    \endgroup
}
```

Again, we could simply copy these lines to our `survey.tex` file.

112 Creating Plots

This section describes the `plot` module.

```
\usepgfmodule{plot}  % LaTeX and plain TeX when using pure PGF
\usepgfmodule[plot]  % ConTeXt when using pure PGF
```

This module provides a set of commands that are intended to make it reasonably easy to plot functions using PGF. It is loaded automatically by `pgf`, but you can load it manually if you have only included `pgfcore`.

112.1 Overview

There are different reasons for using PGF for creating plots rather than some more powerful program such as GNUPLOT or MATHEMATICA, as discussed in Section 22.1. So, let us assume that – for whatever reason – you wish to use PGF for generating a plot.

PGF (conceptually) uses a two-stage process for generating plots. First, a *plot stream* must be produced. This stream consists (more or less) of a large number of coordinates. Second a *plot handler* is applied to the stream. A plot handler "does something" with the stream. The standard handler will issue line-to operations to the coordinates in the stream. However, a handler might also try to issue appropriate curve-to operations in order to smooth the curve. A handler may even do something else entirely, like writing each coordinate to another stream, thereby duplicating the original stream.

Both for the creation of streams and the handling of streams different sets of commands exist. The commands for creating streams start with `\pgfplotstream`, the commands for setting the handler start with `\pgfplothandler`.

112.2 Generating Plot Streams

112.2.1 Basic Building Blocks of Plot Streams

A *plot stream* is a (long) sequence of the following commands:

1. `\pgfplotstreamstart`,
2. `\pgfplotstreampoint`,
3. `\pgfplotstreampointoutlier`,
4. `\pgfplotstreampointundefined`,
5. `\pgfplotstreamnewdataset`,
6. `\pgfplotstreamspecial`, and
7. `\pgfplotstreamend`.

Between calls of these commands arbitrary other code may be called. Obviously, the stream should start with the first command and end with the last command. Here is an example of a plot stream:

```
\pgfplotstreamstart
\pgfplotstreampoint{\pgfpoint{1cm}{1cm}}
\newdimen\mydim
\mydim=2cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\advance \mydim by 3cm
\pgfplotstreampoint{\pgfpoint{\mydim}{2cm}}
\pgfplotstreamend
```

Streams are *global*, meaning that they are not influenced by TeX groups.

`\pgfplotstreamstart`

This command signals that a plot stream starts. The effect of this command is to call the internal command `\pgf@plotstreamstart`, which is set by the current plot handler to do whatever needs to be done at the beginning of the plot. It will also reset the meaning of the internal commands like `\pgf@plotstreampoint` to the initial setting for the plot handler (what this means will be explained in a moment).

\pgfplotstreampoint{⟨*point*⟩}

This command adds a ⟨*point*⟩ to the current plot stream. The effect of this command is to call the internal command `\pgf@plotstreampoint`, which is also set by the current plot handler. This command should now "handle" the point in some sensible way. For example, a line-to command might be issued for the point.

When a plot handler is installed, it will setup the internal command `\pgf@plotstreampoint` in some way. It is permissible to change the meaning of this internal command during a stream. For instance, a handler might setup `\pgf@plotstreampoint` in some sensible way for the first point and then redefine it so that subsequent points are handled in some other way.

As mentioned earlier, the `\pgfplotstreamstart` will always reset the definition of the internal command to the initial meaning it had when the handler was installed. This is true for the other commands mentioned in the following.

\pgfplotstreampointoutlier{⟨*point*⟩}

An *outlier* is a point that is "out of bounds" in some way. For instance, it might have very large coordinates or the coordinates might just be outside some specified range. Nevertheless, an outlier is still a well-defined point. This command is issued, for instance, by GNUPLOT when a value is outside the specified range.

You can configure how outliers are treated using the following key:

/pgf/handle outlier points in plots=⟨*how*⟩ (no default, initially `jump`)

alias `/tikz/handle outlier points in plots`

You can set ⟨*how*⟩ to one of the following values:

- `plot` This will cause the outlier to be drawn normally, just as if `\pgfplotstreampoint` had been used rather than this command.
- `ignore` The outlier will be completely ignored, just as if the command had not been used at all.
- `jump` This causes the internal macro `\pgf@plotstreamjump` to be called. A "jump" in a stream is a position where a "gap" is introduced. For instance, a simple line-to plot handler will stop the current subpath at a jump position and begin with a move-to operation at the next normal point of the stream.
  The net effect of this setting is that at outlier points plots get interrupted and "restarted" when the points are no longer outliers. This is usually the behavior you will be looking for.

\pgfplotstreampointundefined

This command indicated that the stream contains an "undefined" point like a point where some coordinate results for a division by zero. Such a point cannot be plotted, which is why it is not given as a parameter. However, such a point *can* result in a jump in the plot, depending on the setting of the following key:

/pgf/handle undefined points in plots=⟨*how*⟩ (no default, initially `jump`)

alias `/tikz/handle undefined points in plots`

You can set ⟨*how*⟩ to one of the following values:

- `ignore` The undefined point will be completely ignored, just as if the command had not been used at all.
- `jump` This causes the internal macro `\pgf@plotstreamjump` to be called.

\pgfplotstreamnewdataset

This command indicated that in the stream a "new data set" starts. So, the stream does not end, but there is a logical break in the data. For example, when a table is read from a file, empty lines are interpreted as indicating new data sets. What happens when a new data set is encountered is governed by the following key:

/pgf/handle new data sets in plots=⟨*how*⟩ (no default, initially `jump`)

alias `/tikz/handle new data sets in plots`

You can set ⟨*how*⟩ to one of the following values:

- `ignore` The command will be completely ignored, just as if the command had not been used at all.

- `jump` This causes the internal macro `\pgf@plotstreamjump` to be called.

`\pgfplotstreamspecial{`⟨*text*⟩`}`

This command causes `\pgf@plotstreamspecial` to be called with ⟨*text*⟩ as its parameter. This allows handler-specific information to be passed to the handler. All normal handlers ignore this command.

`\pgfplotstreamend`

This command signals that a plot stream ends. It calls `\pgf@plotstreamend`, which should now do any necessary "cleanup".

Note that plot streams are not buffered, that is, the different points are handled immediately. However, using the recording handler, it is possible to record a stream.

112.2.2 Commands That Generate Plot Streams

Plot streams can be created "by hand" as in the earlier example. However, most of the time the coordinates will be produced internally by some command. For example, the `\pgfplotxyfile` reads a file and converts it into a plot stream.

`\pgfplotxyfile{`⟨*filename*⟩`}`

This command will try to open the file ⟨*filename*⟩. If this succeeds, it will convert the file contents into a plot stream as follows: A `\pgfplotstreamstart` is issued. Then, for each empty line a `\pgfplotstreamnewdataset` is produced. Other lines in the file should start with two numbers separated by a space, such as `0.1 1` or `100 -.3`. The numbers may be followed by some text, which will be ignore *except* if it is exactly "`u`" or "`o`". For "`u`" the point is considered to be undefined and `\pgfplotstreampointundefined` is called. For "`o`" the point is considered to be an outlier and `\pgfplotstreampointoutlier` is called. Otherwise, each pair ⟨x⟩ and ⟨y⟩ of numbers is converted into one plot stream point in the xy-coordinate system. Thus, a line like

```
0 Nan u
1 1 some text
2 4
3 9

4 16 o
5 25 oo
```

is turned into

```
\pgfplotstreamstart
\pgfplotstreampointundefined
\pgfplotstreampoint{\pgfpointxy{1}{1}}
\pgfplotstreampoint{\pgfpointxy{2}{4}}
\pgfplotstreampoint{\pgfpointxy{3}{9}}
\pgfplotstreamnewdataset
\pgfplotstreampointoutlier{\pgfpointxy{4}{16}}
\pgfplotstreampoint{\pgfpointxy{5}{25}}
\pgfplotstreamend
```

(Note that the last line is not an outlier because `oo` is not the same as `o`).

The two characters `%` and `#` are also allowed in a file and they are both treated as comment characters. Thus, a line starting with either of them is treated as empty.

When the file has been read completely, `\pgfplotstreamend` is called.

`\pgfplotxyzfile{`⟨*filename*⟩`}`

This command works like `\pgfplotxyfile`, only *three* numbers are expected on each non-empty line. They are converted into points in the xyz-coordinate system. Consider, the following file:

```
% Some comments
# more comments
2 -5  1 first entry
2 -.2 2 o
2 -5  2 third entry
```

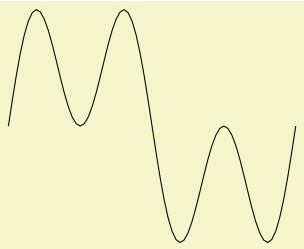
It is turned into the following stream:

```
\pgfplotstreamstart
\pgfplotstreamnewdataset
\pgfplotstreamnewdataset
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{1}}
\pgfplotstreampointoutlier{\pgfpointxyz{2}{-.2}{2}}
\pgfplotstreampoint{\pgfpointxyz{2}{-5}{2}}
\pgfplotstreamend
```

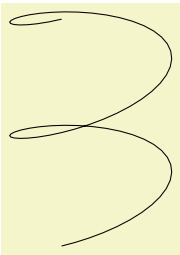
Currently, there is no command that can decide automatically whether the xy-coordinate system should be used or whether the xyz-system should be used. However, it would not be terribly difficult to write a "smart file reader" that parses coordinate files a bit more intelligently.

\pgfplotfunction{⟨*variable*⟩}{⟨*sample list*⟩}{⟨*point*⟩}

This command will produce coordinates by iterating the ⟨*variable*⟩ over all values in ⟨*sample list*⟩, which should be a list in the `\foreach` syntax. For each value of ⟨*variable*⟩, the ⟨*point*⟩ is evaluated and the resulting coordinate is inserted into the plot stream.



```
\begin{tikzpicture}[x=3.8cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\x}{0,5,...,360}{\pgfpointxy{\x}{sin(\x)+sin(3*\x)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```



```
\begin{tikzpicture}[y=3cm/360]
  \pgfplothandlerlineto
  \pgfplotfunction{\y}{0,5,...,360}{\pgfpointxyz{sin(2*\y)}{\y}{cos(2*\y)}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

Be warned that if the expressions that need to evaluated for each point are complex, then this command can be very slow.

\pgfplotgnuplot[⟨*prefix*⟩]{⟨*function*⟩}

This command will "try" to call the GNUPLOT program to generate the coordinates of the ⟨*function*⟩. In detail, the following happens:

This command works with two files: ⟨*prefix*⟩`.gnuplot` and ⟨*prefix*⟩`.table`. If the optional argument ⟨*prefix*⟩ is not given, it is set to `\jobname`.

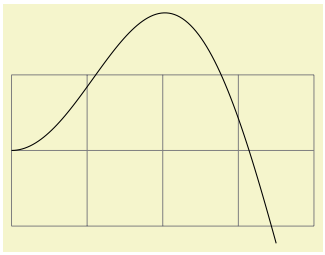
Let us start with the situation where none of these files exists. Then PGF will first generate the file ⟨*prefix*⟩`.gnuplot`. In this file it writes

```
set table "#1.table"; set format "%.5f"
```

where `#1` is replaced by ⟨*prefix*⟩. Then, in a second line, it writes the text ⟨*function*⟩.

Next, PGF will try to invoke the program `gnuplot` with the argument ⟨*prefix*⟩`.gnuplot`. This call may or may not succeed, depending on whether the `\write18` mechanism (also known as shell escape) is switched on and whether the `gnuplot` program is available.

Assuming that the call succeeded, the next step is to invoke `\pgfplotxyfile` on the file ⟨*prefix*⟩`.table`; which is exactly the file that has just been created by `gnuplot`.



```
\begin{tikzpicture}
  \draw[help lines] (0,-1) grid (4,1);
  \pgfplothandlerlineto
  \pgfplotgnuplot[plots/pgfplotgnuplot-example]{plot [x=0:3.5] x*sin(x)}
  \pgfusepath{stroke}
\end{tikzpicture}
```

The more difficult situation arises when the `.gnuplot` file exists, which will be the case on the second run of TEX on the TEX file. In this case PGF will read this file and check whether it contains exactly what PGF "would have written" into this file. If this is not the case, the file contents is overwritten with what "should be there" and, as above, `gnuplot` is invoked to generate a new `.table` file. However, if the file contents is "as expected", the external `gnuplot` program is *not* called. Instead, the ⟨*prefix*⟩`.table` file is immediately read.

As explained in Section 22.6, the net effect of the above mechanism is that `gnuplot` is called as seldom as possible and that when you pass along the `.gnuplot` and `.table` files with your `.tex` file to someone else, that person can TEX the `.tex` file without having `gnuplot` installed and without having the `\write18` mechanism switched on.

`/pgf/plot/gnuplot call=`⟨*gnuplot invocation*⟩ (no default, initially `gnuplot`)

This key can be used to change the way gnuplot is called.

Some portable MiKTEX distribution needs something like the following.

```
\pgfkeys{/pgf/plot/gnuplot call="/Programs/gnuplot/binary/gnuplot"}
```

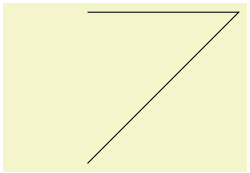
112.3 Plot Handlers

A *plot handler* determines what "should be done" with a plot stream. You must set the plot handler before the stream starts. The following commands install the most basic plot handlers; more plot handlers are defined in the file `pgflibraryplothandlers`, which is documented in Section 65.

All plot handlers work by setting or redefining the following three macros: `\pgf@plotstreamstart`, `\pgf@plotstreampoint`, and `\pgf@plotstreamend`.

`\pgfplothandlerlineto`

This handler will issue a `\pgfpathlineto` command for each point of the plot, *except* possibly for the first. What happens with the first point can be specified using the two commands described below.



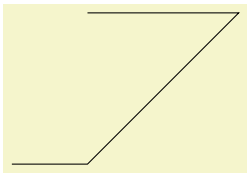
```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

`\pgfsetmovetofirstplotpoint`

Specifies that the line-to plot handler (and also some other plot handlers) should issue a move-to command for the first point of the plot instead of a line-to. This will start a new part of the current path, which is not always, but often, desirable. This is the default.

`\pgfsetlinetofirstplotpoint`

Specifies that plot handlers should issue a line-to command for the first point of the plot.



```
\begin{pgfpicture}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfsetlinetofirstplotpoint
  \pgfplothandlerlineto
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{2cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

\pgfplothandlerpolygon

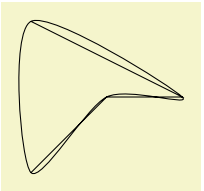
This handler works like the line-to plot handler, only the line is closed at the end using `\pgfpathclose`, resulting in a polygon.

\pgfplothandlerdiscard

This handler will simply throw away the stream.

\pgfplothandlerrecord{⟨*macro*⟩}

When this handler is installed, each time a plot stream command is called, this command will be appended to ⟨*macro*⟩. Thus, at the end of the stream, ⟨*macro*⟩ will contain all the commands that were issued on the stream. You can then install another handler and invoke ⟨*macro*⟩ to "replay" the stream (possibly many times).



```
\begin{pgfpicture}
  \pgfplothandlerrecord{\mystream}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{1cm}{0cm}}
  \pgfplotstreampoint{\pgfpoint{2cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{3cm}{1cm}}
  \pgfplotstreampoint{\pgfpoint{1cm}{2cm}}
  \pgfplotstreamend
  \pgfplothandlerlineto
  \mystream
  \pgfplothandlerclosedcurve
  \mystream
  \pgfusepath{stroke}
\end{pgfpicture}
```

112.4 Defining New Plot Handlers

You can define new plot handlers using the following command:

\pgfdeclareplothandler{⟨*macro*⟩}{⟨*arguments*⟩}{⟨*configuration*⟩}

This command creates a new plot handler that can subsequently be called using the macro ⟨*macro*⟩. This macro take the arguments given in ⟨*arguments*⟩, which can be a list like `#1#2` if ⟨*macro*⟩ should be invoked with two arguments. Here is a typical example:

```
\pgfdeclareplothandler{\myhandler}{#1}{...}
...
\myhandler{foo}
\pgfplotstreamstart
...
\pgfplotstreamend
```

The ⟨*configuration*⟩ is used to define the behavior of the handler. It is a list of key–value pairs, where the following keys are allowed:

- `start=`⟨*code*⟩. The ⟨*code*⟩ will be executed whenever `\pgfplotstreamstart` is used while the handler ⟨*macro*⟩ is selected. Inside the ⟨*code*⟩, you can use `#1`, `#2`, and so on to refer to the parameters that were given to ⟨*macro*⟩:

Hi foo.Bye foo.Hi bar.Bye bar.

```
\pgfdeclareplothandler{\myhandler}{#1}{
  start = Hi #1.,
  end   = Bye #1.,
}
\myhandler{foo}
\pgfplotstreamstart
\pgfplotstreamend
\myhandler{bar}
\pgfplotstreamstart
\pgfplotstreamend
```

- `end=`⟨*code*⟩ Works just like `start`.
- `point=`⟨*code*⟩. The ⟨*code*⟩ will be executed whenever `\pgfplotstreampoint` is used while the handler ⟨*macro*⟩ is in force. Inside the ⟨*code*⟩, you can use `#1`, `#2`, and so on to refer to the arguments give to ⟨*macro*⟩, while you can use `##1` to refer to the argument given to `\pgfplotstreampoint` itself (this will be the coordinate).



```
\pgfdeclareplothandler{\myhandler}{#1}{
  point=\pgfpathcircle{##1}{#1} % ##1 is the coordinate,
                                % #1 the parameter for \myhandler
}
\begin{pgfpicture}
  \myhandler{1pt}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0pt}{0pt}}
  \pgfplotstreampoint{\pgfpoint{3pt}{3pt}}
  \pgfplotstreampoint{\pgfpoint{6pt}{3pt}}
  \pgfplotstreampoint{\pgfpoint{9pt}{0pt}}
  \pgfplotstreamend
  \pgfusepath{stroke}
  \myhandler{3pt}
  \pgfplotstreamstart
  \pgfplotstreampoint{\pgfpoint{0pt}{0pt}}
  \pgfplotstreampoint{\pgfpoint{9pt}{0pt}}
  \pgfplotstreamend
  \pgfusepath{stroke}
\end{pgfpicture}
```

  The ⟨*code*⟩ will also be called for `\pgfplotstreampointoutlier` when this command has been configured to `plot` the outliers.
- `jump=`⟨*code*⟩ The ⟨*code*⟩ will be called whenever a jump has been requested indirectly via an outlier point, and undefined point, or a new data set (for each of which the command needs to be configured to `jump`). As always, inside the ⟨*code*⟩ you can access `#1` and so on.
- `special=`⟨*code*⟩ Causes ⟨*code*⟩ to be executed whenever `\pgfplotstreamspecial{`⟨*something*⟩`}` is used. Inside the ⟨*code*⟩, you can access ⟨*something*⟩ via `##1` and the parameters of ⟨*macro*⟩ as `#1`, `#2`, and so on.

In addition to the above keys, there exist also "code macro versions" of them:

- `point macro=`⟨*some macro*⟩. Causes `\pgfplotstreampoint` to call ⟨*some macro*⟩ directly (actually, `\pgf@plotstreampoint` is set to be equal to ⟨*some macro*⟩). Inside the ⟨*some macro*⟩ you can use `#1` to access the coordinate passed to `\pgfplotstreampoint` and you can no longer access the parameters passed to the original call to ⟨*macro*⟩ that installed the handler. So, ⟨*some macro*⟩ must take exactly one argument, namely `#1`.
- `special macro=`⟨*some macro*⟩. As `point macro`, only for specials.
- `start macro=`⟨*some macro*⟩. Causes ⟨*some macro*⟩ to be executed at the start. This macro, like the below ones, may not take any parameters and will not have access to the parameters passed to the original ⟨*macro*⟩.
- `end macro=`⟨*some macro*⟩. As above.
- `jump macro=`⟨*some macro*⟩. As above.

113 Layered Graphics

113.1 Overview

PGF provides a layering mechanism for composing graphics from multiple layers. (This mechanism is not to be confused with the conceptual "software layers" the PGF system is composed of.) Layers are often used in graphic programs. The idea is that you can draw on the different layers in any order. So you might start drawing something on the "background" layer, then something on the "foreground" layer, then something on the "middle" layer, and then something on the background layer once more, and so on. At the end, no matter in which ordering you drew on the different layers, the layers are "stacked on top of each other" in a fixed ordering to produce the final picture. Thus, anything drawn on the middle layer would come on top of everything of the background layer.

Normally, you do not need to use different layers since you will have little trouble "ordering" your graphic commands in such a way that layers are superfluous. However, in certain situations you only "know" what you should draw behind something else after the "something else" has been drawn.

For example, suppose you wish to draw a yellow background behind your picture. The background should be as large as the bounding box of the picture, plus a little border. If you know the size of the bounding box of the picture at its beginning, this is easy to accomplish. However, in general this is not the case and you need to create a "background" layer in addition to the standard "main" layer. Then, at the end of the picture, when the bounding box has been established, you can add a rectangle of the appropriate size to the picture.

113.2 Declaring Layers

In PGF layers are referenced using names. The standard layer, which is a bit special in certain ways, is called `main`. If nothing else is specified, all graphic commands are added to the `main` layer. You can declare a new layer using the following command:

`\pgfdeclarelayer{`⟨*name*⟩`}`

This command declares a layer named ⟨*name*⟩ for later use. Mainly, this will set up some internal bookkeeping.

The next step toward using a layer is to tell PGF which layers will be part of the actual picture and which will be their ordering. Thus, it is possible to have more layers declared than are actually used.

`\pgfsetlayers{`⟨*layer list*⟩`}`

This command tells PGF which layers will be used in pictures. They are stacked on top of each other in the order given. The layer `main` should always be part of the list. Here is an example:

```
\pgfdeclarelayer{background}
\pgfdeclarelayer{foreground}
\pgfsetlayers{background,main,foreground}
```

This command should be given either outside of any picture or "directly inside" of a picture. Here, the "directly inside" means that there should be no further level of TEX grouping between `\pgfsetlayers` and the matching `\end{pgfpicture}` (no closing braces, no `\end{...}`). It will also work if `\pgfsetlayers` is provided before `\end{tikzpicture}` (with similar restrictions).

113.3 Using Layers

Once the layers of your picture have been declared, you can start to "fill" them. As said before, all graphics commands are normally added to the `main` layer. Using the `{pgfonlayer}` environment, you can tell PGF that certain commands should, instead, be added to the given layer.

`\begin{pgfonlayer}{`⟨*layer name*⟩`}`
⟨*environment contents*⟩
`\end{pgfonlayer}`

The whole ⟨*environment contents*⟩ is added to the layer with the name ⟨*layer name*⟩. This environment can be used anywhere inside a picture. Thus, even if it is used inside a `{pgfscope}` or a TEX group, the contents will still be added to the "whole" picture. Using this environment multiple times inside the same picture will cause the ⟨*environment contents*⟩ to accumulate.

Note: You can *not* add anything to the `main` layer using this environment. The only way to add anything to the main layer is to give graphic commands outside all `{pgfonlayer}` environments.




```
\pgfdeclarelayer{background layer}
\pgfdeclarelayer{foreground layer}
\pgfsetlayers{background layer,main,foreground layer}
\begin{tikzpicture}
  % On main layer:
  \fill[blue] (0,0) circle [radius=1cm];

  \begin{pgfonlayer}{background layer}
    \fill[yellow] (-1,-1) rectangle (1,1);
  \end{pgfonlayer}

  \begin{pgfonlayer}{foreground layer}
    \node[white] {foreground};
  \end{pgfonlayer}

  \begin{pgfonlayer}{background layer}
    \fill[black] (-.8,-.8) rectangle (.8,.8);
  \end{pgfonlayer}

  % On main layer again:
  \fill[blue!50] (-.5,-1) rectangle (.5,1);
\end{tikzpicture}
```

`\pgfonlayer{`⟨*layer name*⟩`}`
⟨*environment contents*⟩
`\endpgfonlayer`

The plain TEX version of the environment.

`\startpgfonlayer{`⟨*layer name*⟩`}`
⟨*environment contents*⟩
`\stoppgfonlayer`

The ConTEXt version of the environment.

114 Shadings

114.1 Overview

A shading is an area in which the color changes smoothly between different colors. Similarly to an image, a shading must first be declared before it can be used. Also similarly to an image, a shading is put into a TEX-box. Hence, in order to include a shading in a `{pgfpicture}`, you have to use `\pgftext` around it.

There are different kinds of shadings: horizontal, vertical, radial, and functional shadings. However, you can rotate and clip shadings like any other graphics object, which allows you to create more complicated shadings. Horizontal shadings could be created by rotating a vertical shading by 90 degrees, but explicit commands for creating both horizontal and vertical shadings are included for convenience.

Once you have declared a shading, you can insert it into the text using the command `\pgfuseshading`. This command cannot be used directly in a `{pgfpicture}`, you have to put a `\pgftext` around it. The second command for using shadings, `\pgfshadepath`, on the other hand, can only be used inside `{pgfpicture}` environments. It will "fill" the current path with the shading.

A horizontal shading is a horizontal bar of a certain height whose color changes smoothly. You must at least specify the colors at the left and at the right end of the bar, but you can also add color specifications for points in between. For example, suppose you wish to create a bar that is red at the left end, green in the middle, and blue at the end, and you would like the bar to be 4cm long. This could be specified as follows:

```
rgb(0cm)=(1,0,0); rgb(2cm)=(0,1,0); rgb(4cm)=(0,0,1)
```

This line means that at 0cm (the left end) of the bar, the color should be red, which has red-green-blue (rgb) components (1,0,0). At 2cm, the bar should be green, and at 4cm it should be blue. Instead of `rgb`, you can currently also specify `cmyk` as color model, in which case four values are needed, `gray` as color model, in which case only one value is needed, or `color`, in which case you must provide the name of a color in parentheses. In a color specification the individual specifications must be separated using a semicolon, which may be followed by a whitespace (like a space or a newline). Individual specifications must be given in increasing order.

114.1.1 Color models

by David Purton

An attempt is made to produce shadings consistent with the currently selected `xcolor` package color model. The `rgb`, `cmyk`, and `gray` color models from the `xcolor` package are supported.

Note: The color model chosen for a shading is based on the `xcolor` color model *at the time the shading is created.* This is either when `\pgfdeclare*shading` is called with no optional argument or when `\pgfuseshading` is called if `\pgfdeclare*shading` was called with an optional argument.

If the `xcolor` package `natural` color model is in use then the shading color model will be RGB by default. In practice this means that if you are using the `natural` color model of the `xcolor` package you can get mismatched colors if you, for example, create a shading from green (which is defined as RGB) to magenta (which is defined as CMYK). The shading will finish with RGB magenta which will look different to the CMYK magenta used in solid colors.

You can avoid mismatched colors by loading the `xcolor` package first with an explicit color model (`rgb`, `cmyk`, or `gray`).

```
\begin{tikzpicture}
  \fill[green] (0,0) rectangle (1,1);
  \shade[left color=green, right color=magenta] (1.25,0) rectangle (3.75,1);
  \fill[magenta] (4,0) rectangle (5,1);
\end{tikzpicture}
```

`xcolor natural` color model:



`xcolor cmyk` color model:



`xcolor rgb` color model:



`xcolor gray` color model:



114.2 Declaring Shadings

114.2.1 Horizontal and Vertical Shadings

`\pgfdeclarehorizontalshading`[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*shading height*⟩}{⟨*color specification*⟩}

Declares a horizontal shading named ⟨*shading name*⟩ of the specified ⟨*height*⟩ with the specified colors. The width of the bar is deduced automatically from the maximum dimension in the specification.



```
\pgfdeclarehorizontalshading{myshadingA}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(green); color(4cm)=(blue)}
\pgfuseshading{myshadingA}
```

The effect of the ⟨*color list*⟩, which is a comma-separated list of colors, is the following: Normally, when this list is empty, once a shading has been declared, it becomes "frozen". This means that even if you change a color that was used in the declaration of the shading later on, the shading will not change. By specifying a ⟨*color list*⟩ you can specify that the shading should be recalculated whenever one of the colors listed in the list changes (this includes effects like color mixins and `xcolor` color models). Thus, when you specify a ⟨*color list*⟩, whenever the shading is used, PGF first converts the colors in the list to tuples in the current `xcolor` color model using the current values of the colors and taking any mixins and blends into account. If the resulting tuples have not yet been used, a new shading is internally created and used. Note that if the option ⟨*color list*⟩ is used, then no shading is created until the first use of `\pgfuseshading`. In particular, the colors mentioned in the shading need not be defined when the declaration is given.

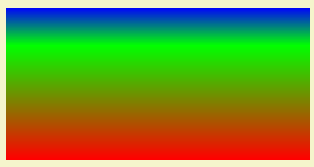
When a shading is recalculated because of a change in the colors mentioned in ⟨*color list*⟩, the complete shading is recalculated. Thus even colors not mentioned in the list will be used with their current values, not with the values they had upon declaration.



```
\pgfdeclarehorizontalshading[mycolor]{myshadingB}
  {1cm}{rgb(0cm)=(1,0,0); color(2cm)=(mycolor)}
\colorlet{mycolor}{green}
\pgfuseshading{myshadingB}
\colorlet{mycolor}{blue}
\pgfuseshading{myshadingB}
```

`\pgfdeclareverticalshading`[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*shading width*⟩}{⟨*color specification*⟩}

Declares a vertical shading named ⟨*shading name*⟩ of the specified ⟨*width*⟩. The height of the bar is deduced automatically. The effect of ⟨*color list*⟩ is the same as for horizontal shadings.

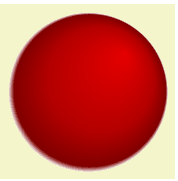


```
\pgfdeclareverticalshading{myshadingC}
  {4cm}{rgb(0cm)=(1,0,0); rgb(1.5cm)=(0,1,0); rgb(2cm)=(0,0,1)}
\pgfuseshading{myshadingC}
```

114.2.2 Radial Shadings

`\pgfdeclareradialshading`[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*center point*⟩}{⟨*color specification*⟩}

Declares a radial shading. A radial shading is a circle whose inner color changes as specified by the color specification. Assuming that the center of the shading is at the origin, the color of the center will be the color specified for 0cm and the color of the border of the circle will be the color for the maximum dimension given in the ⟨*color specified*⟩. This maximum will also be the radius of the circle. If the ⟨*center point*⟩ is not at the origin, the whole shading inside the circle (whose size remains exactly the same) will be distorted such that the given center now has the color specified for 0cm. The effect of ⟨*color list*⟩ is the same as for horizontal shadings.



```
\pgfdeclareradialshading{sphere}{\pgfpoint{0.5cm}{0.5cm}}%
  {rgb(0cm)=(0.9,0,0);
   rgb(0.7cm)=(0.7,0,0);
   rgb(1cm)=(0.5,0,0);
   rgb(1.05cm)=(1,1,1)}
\pgfuseshading{sphere}
```

114.2.3 General (Functional) Shadings

\pgfdeclarefunctionalshading[⟨*color list*⟩]{⟨*shading name*⟩}{⟨*lower left corner*⟩}{⟨*upper right corner*⟩}{⟨*init code*⟩}{⟨*type 4 function*⟩}

Warning: These shadings are the least portable of all and they put the heaviest burden of the renderer. They are slow and, possibly, will not print correctly!

This command creates a *functional shading*. For such a shading, the color of each point is calculated by calling a function that takes the coordinates of the point as input and yields the color as an output. Note that the function is evaluated by the *renderer*, not by PGF or TEX or someone else at compile-time. This means that the evaluation of this function has to be done *extremely quickly* and the function should be *very simple*. For this reason, only a very restricted set of operations are possible in the function and functions should be kept small. Any errors in the function will only be noticed by the renderer.

The syntax for specifying functions is the following: You use a simplified form of a subset of the PostScript language. This subset will be understood by the PDF-renderer (yes, PDF-renderers do have a basic understanding of PostScript) and also by PostScript renders. This subset is detailed in Section 3.9.4 of the PDF-specification (version 1.7). In essence, the specification states that these functions may contain "expressions involving integers, real numbers, and boolean values only. There are no composite data structures such as strings or arrays, no procedures, and no variables or names." The allowed operators are (exactly) the following: `abs`, `add`, `atan`, `ceiling`, `cos`, `cvi`, `cvr`, `div`, `exp`, `floor`, `idiv`, `ln`, `log`, `mod`, `mul`, `neg`, `round`, `sin`, `sqrt`, `sub`, `truncate`, `and`, `bitshift`, `eq`, `false`, `ge`, `gt`, `le`, `lt`, `ne`, `not`, `or`, `true`, `xor`, `if`, `ifelse`, `copy`, `dup`, `exch`, `index`, `pop`, `roll`.

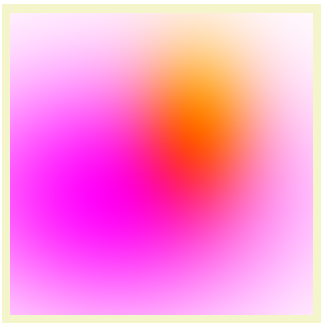
When the function is evaluated, the top two stack elements are the coordinates of the point for which the color should be computed. The coordinates are dimensionless and given in big points, so for the coordinate $(50bp, 72.27pt)$ the top two stack elements would be `50.0` and `72.0`. Otherwise, the (virtual) stack is empty (or should be treated as if it were empty). The function should then replace these two values by three values, representing the red, green, and blue color of the point for an RGB shading, four colors, representing the cyan, magenta, yellow, and black color of the point for a CMYK shading, or one value representing the gray color for a grayscale shading. The numbers should be real values, not integers since, Apple's PDF renderer is broken in this regard (use `cvr` at the end if necessary).

Conceptually, the function will be evaluated once for each point of the rectangle ⟨*lower left corner*⟩ to ⟨*upper right corner*⟩, which should be a PGF-point expression like `\pgfpoint{100bp}{100bp}`. A renderer may choose to evaluate the function at less points, but, in principle, the function will be evaluated for each pixel independently.

Because of the rather difficult PostScript syntax, use this macro only *if you know what you are doing* (or if you are adventurous, of course).

As for other shadings, the optional ⟨*color list*⟩ is used to determine whether a shading needs to be recalculated when a color has changed.

The ⟨*init code*⟩ is executed each time a shading is (re)calculated. Typically, it will contain code to extract components from colors.



```
\pgfdeclarefunctionalshading{twospots}
    {\pgfpointorigin}{\pgfpoint{4cm}{4cm}}{}{
  % Save coordinates for later
  2 copy
  % Compute distance from (40bp,45bp), with x doubled
  45 sub dup mul exch
  40 sub dup mul 0.5 mul add sqrt
  % exponential decay
  dup mul neg 1.0005 exch exp 1.0 exch sub
  % Compute distance from (70bp,70bp) from stored coordinate, scaled
  3 1 roll
  70 sub dup mul .5 mul exch
  70 sub dup mul add sqrt
  % Decay
  dup mul neg 1.002 exch exp 1.0 exch sub
  % red component
  1.0 3 1 roll
}
\pgfuseshading{twospots}
```

Inside the PostScript function ⟨*type 4 function*⟩ you cannot use colors directly. Rather, you must push the color components on the stack. For this, it is useful to call one of `\pgfshadecolortorgb`, `\pgfshadecolortocmyk`, or `\pgfshadecolortogray` in the ⟨*init code*⟩:

`\pgfshadecolortorgb{`⟨*color name*⟩`}{`⟨*macro*⟩`}`

This command takes ⟨*color name*⟩ as input, converts it to RGB and stores the color's red/green/blue components real numbers between 0.0 and 1.0 separated by spaces (which is exactly what you need if you want to push it on a stack) in ⟨*macro*⟩. This macro can then be used inside the ⟨*type 4 function*⟩ argument for `\pgfdeclarefunctionalshading`.

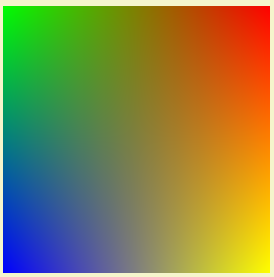


```
\pgfdeclarefunctionalshading[mycol]{sweep}{\pgfpoint{-1cm}{-1cm}}
{\pgfpoint{1cm}{1cm}}{\pgfshadecolortorgb{mycol}{\myrgb}}{
  2 copy        % whirl
  % Calculate "safe" atan of position
  2 copy abs exch abs add 0.0001 ge { atan } { pop } ifelse
  3 1 roll
  dup mul exch
  dup mul add sqrt
  30 mul
  add
  sin
  1 add 2 div
  dup
  \myrgb        % push mycol
  5 4 roll      % multiply all components by calculated value
  mul
  3 1 roll
  3 index
  mul
  3 1 roll
  4 3 roll
  mul
  3 1 roll
}
\colorlet{mycol}{white}%
\pgfuseshading{sweep}%
\colorlet{mycol}{red}%
\pgfuseshading{sweep}
```

In addition, three macros suffixed with `red`, `green` and `blue` are defined, which store the individual components of ⟨*color name*⟩. These can also be used in the ⟨*type 4 function*⟩ argument.

`\mycol`=1.0 0.5 0.0 `\mycolred`=1.0 `\mycolgreen`=0.5 `\mycolblue`=0.0

```
\usepackage {shortvrb} \MakeShortVerb {\|}
\pgfshadecolortorgb{orange}{\mycol}
|\mycol|=\mycol |\mycolred|=\mycolred |\mycolgreen|=\mycolgreen |\mycolblue|=\mycolblue
```



```
\pgfdeclarefunctionalshading[col1,col2,col3,col4]{bilinear interpolation}
{\pgfpointorigin}{\pgfpoint{100bp}{100bp}}
{
\pgfshadecolortorgb{col1}{\first}\pgfshadecolortorgb{col2}{\second}
\pgfshadecolortorgb{col3}{\third}\pgfshadecolortorgb{col4}{\fourth}
}{
  100 div exch 100 div 2 copy                    % Calculate y/100 x/100.
  neg 1 add exch neg 1 add                       % Calculate 1-y/100 1-x/100.
  3 1 roll 2 copy exch 5 2 roll 6 copy 6 copy    % Set up stack.
  \firstred mul exch \secondred mul add mul      % Process red component.
  4 1 roll
  \thirdred mul exch \fourthred mul add mul
  add
  13 1 roll
  \firstgreen mul exch \secondgreen mul add mul % Process green component.
  4 1 roll
  \thirdgreen mul exch \fourthgreen mul add mul
  add
  7 1 roll
  \firstblue mul exch \secondblue mul add mul    % Process blue component.
  4 1 roll
  \thirdblue mul exch \fourthblue mul add mul
  add
}

\colorlet{col1}{blue}
\colorlet{col2}{yellow}
\colorlet{col3}{red}
\colorlet{col4}{green}
\pgfuseshading{bilinear interpolation}
```

\pgfshadecolortocmyk{⟨*color name*⟩}{⟨*macro*⟩}

This command takes ⟨*color name*⟩ as input, converts it to CMYK and stores the color's cyan/magenta/yellow/black components real numbers between 0.0 and 1.0 separated by spaces.

In addition, four macros suffixed with `cyan`, `magenta`, `yellow` and `black` are defined, which store the individual components of ⟨*color name*⟩.

\pgfshadecolortogray{⟨*color name*⟩}{⟨*macro*⟩}

This command takes ⟨*color name*⟩ as input converts it to grayscale and stores the color's value as a real number between 0.0 and 1.0.

Although it's not needed, for consistency a second macro suffixed with `gray` is also defined.

Color model independent functional shadings. By nature, the PostScript code used in functional shadings must output one of RGB, CMYK, or grayscale data. Therefore, `\pgfdeclarefunctionalshading` is *not* portable across color models.

Take particular care that the same color model is in use at declaration time and use time for functional shadings declared with an optional argument as otherwise the PostScript data will not match the declared color space and you will end up with a malformed PDF.

Having said this, it *is* possible to create portable functional shadings by providing conditional code to append color transformations to the PostScript data. A variety of `\pgffuncshading*to*` (e.g., `\pgffuncshadingrgbtocmyk`) macros along with `\ifpgfshadingmodel*` (e.g., `\ifpgfshadingmodelcmyk`) conditionals are provided to assist with these transformations. Obviously, this will make the PostScript code less efficient than if you work in your intended color model.

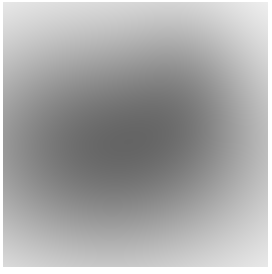
`xcolor rgb` model:



`xcolor cmyk` model:



`xcolor gray` model:



```
\pgfdeclarefunctionalshading[black]{portabletwospots}{\pgfpointorigin}{\pgfpoint{3.5cm}{3.5cm}}{}{
  2 copy
  45 sub dup mul exch
  40 sub dup mul 0.5 mul add sqrt
  dup mul neg 1.0005 exch exp 1.0 exch sub
  3 1 roll
  70 sub dup mul .5 mul exch
  70 sub dup mul add sqrt
  dup mul neg 1.002 exch exp 1.0 exch sub
  1.0 3 1 roll
  \ifpgfshadingmodelcmyk
    \pgffuncshadingrgbtocmyk
  \fi
  \ifpgfshadingmodelgray
    \pgffuncshadingrgbtogray
  \fi
}
```

\pgffuncshadingrgbtocmyk

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top 3 elements on the stack from RGB to CMYK. In combination with the `\ifpgfshadingmodelcmyk` conditional this macro can be used to make functional shading declarations more portable across color models.

\pgffuncshadingrgbtogray

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top 3 elements on the stack from RGB to grayscale. In combination with the `\ifpgfshadingmodelgray` conditional this macro can be used to make functional shading declarations more portable across color models.

\pgffuncshadingcmyktorgb

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top 4 elements on the stack from CMYK to RGB. In combination with the `\ifpgfshadingmodelrgb` conditional this macro can be used to make functional shading declarations more portable across color models.

\pgffuncshadingcmyktogray

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top 4 elements on the stack from CMYK to grayscale. In combination with the `\ifpgfshadingmodelgray` conditional this macro can be used to make functional shading declarations more portable across color models.

\pgffuncshadinggraytorgb

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top element on the stack from grayscale to RGB. In combination with the `\ifpgfshadingmodelrgb` conditional this macro can be used to make functional shading declarations more portable across color models.

\pgffuncshadinggraytocmyk

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to convert the top element on the stack from grayscale to CMYK. In combination with the `\ifpgfshadingmodelcmyk` conditional this macro can be used to make functional shading declarations more portable across color models.

\ifpgfshadingmodelrgb

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to test if the `xcolor` color model is `rgb` *at the time the shading is created*. This can be used to ensure that the data output in the ⟨*type 4 function*⟩ correctly matches the active color model.

\ifpgfshadingmodelcmyk

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to test if the `xcolor` color model is `cmyk` *at the time the shading is created*. This can be used to ensure that the data output in the ⟨*type 4 function*⟩ correctly matches the active color model.

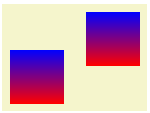
\ifpgfshadingmodelgray

Within the ⟨*type 4 function*⟩ argument of `\pgfdeclarefunctionalshading`, this command can be used to test if the `xcolor` color model is `gray` *at the time the shading is created*. This can be used to ensure that the data output in the ⟨*type 4 function*⟩ correctly matches the active color model.

114.3 Using Shadings

\pgfuseshading{⟨*shading name*⟩}

Inserts a previously declared shading into the text. If you wish to use it in a `pgfpicture` environment, you should put a `\pgftext` around it.



```
\begin{pgfpicture}
  \pgfdeclareverticalshading{myshadingD}
    {20pt}{color(0pt)=(red); color(20pt)=(blue)}
  \pgftext[at=\pgfpoint{1cm}{0cm}]   {\pgfuseshading{myshadingD}}
  \pgftext[at=\pgfpoint{2cm}{0.5cm}]{\pgfuseshading{myshadingD}}
\end{pgfpicture}
```

\pgfshadepath{⟨*shading name*⟩}{⟨*angle*⟩}

This command must be used inside a `{pgfpicture}` environment. The effect is a bit complex, so let us go over it step by step.

First, PGF will set up a local scope.

Second, it uses the current path to clip everything inside this scope. However, the current path is once more available after the scope, so it can be used, for example, to stroke it.

Now, the ⟨*shading name*⟩ should be a shading whose width and height are 100 bp, that is, 100 big points. PGF has a look at the bounding box of the current path. This bounding box is computed automatically when a path is computed; however, it can sometimes be (quite a bit) too large, especially when complicated curves are involved.

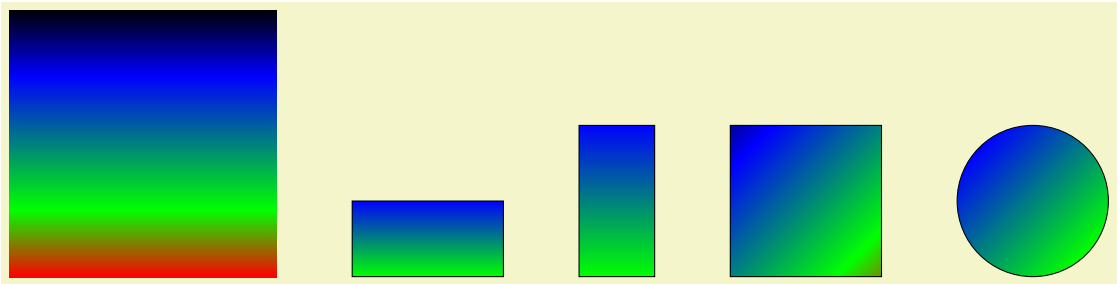
Inside the scope, the low-level transformation matrix is modified. The center of the shading is translated (moved) such that it lies on the center of the bounding box of the path. The low-level coordinate system is also scaled such that the shading "covers" the path (the details are a bit more complex, see below). Then, the coordinate system is rotated by ⟨*angle*⟩. Finally, if the macro `\pgfsetadditionalshadetransform` has been used, an additional transformation is applied.

After everything has been set up, the shading is inserted. Due to the transformations and clippings, the effect will be that the shading seems to "fill" the path.

If both the path and the shadings were always rectangles and if rotations were never involved, it would be easy to scale shadings such they always cover the path. However, when a vertical shading is rotated, it must obviously be "magnified" so that it still covers the path. Things get worse when the path is not a rectangle itself.

For these reasons, things work slightly differently "in reality". The shading is scaled and translated such that the point (50bp, 50bp), which is the middle of the shading, is at the middle of the path and such that the point (25bp, 25bp) is at the lower left corner of the path and that (75bp, 75bp) is at upper right corner.

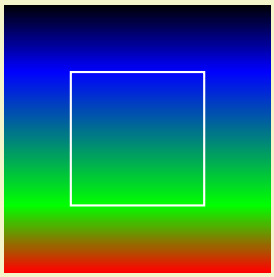
In other words, only the center quarter of the shading will actually "survive the clipping" if the path is a rectangle. If the path is not a rectangle, but, say, a circle, even less is seen of the shading. Here is an example that demonstrates this effect:



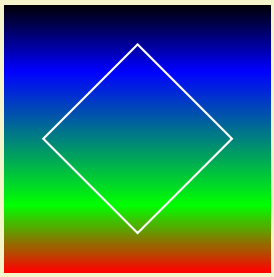
```
\pgfdeclareverticalshading{myshadingE}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\pgfuseshading{myshadingE}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{1cm}{2cm}}
  \pgfshadepath{myshadingE}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{5cm}{0cm}}{\pgfpoint{2cm}{2cm}}
  \pgfshadepath{myshadingE}{45}
  \pgfusepath{stroke}
  \pgfpathcircle{\pgfpoint{9cm}{1cm}}{1cm}
  \pgfshadepath{myshadingE}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

As can be seen above in the last case, the "hidden" part of the shading actually *can* become visible if the shading is rotated. The reason is that it is scaled as if no rotation took place, then the rotation is done.

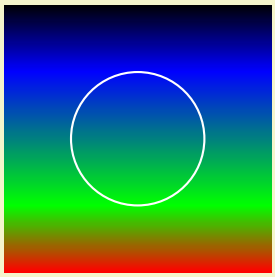
The following graphics show which part of the shading are actually shown:



first two applications



third application



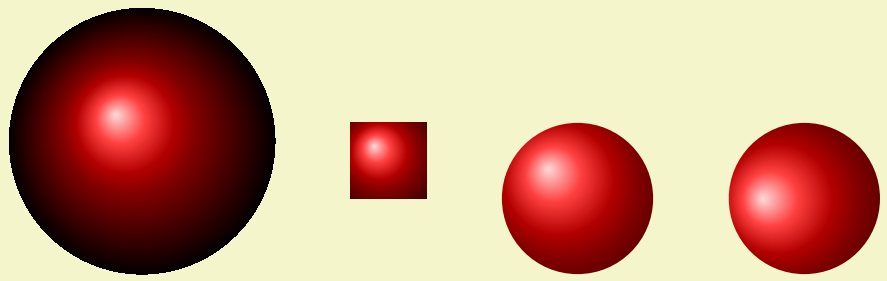
fourth application

```
\pgfdeclareverticalshading{myshadingF}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\begin{tikzpicture}
  \draw (50bp,50bp) node {\pgfuseshading{myshadingF}};
  \draw[white,thick] (25bp,25bp) rectangle (75bp,75bp);
  \draw (50bp,0bp) node[below] {first two applications};

  \begin{scope}[xshift=5cm]
    \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
    \draw[rotate around={45:(50bp,50bp)},white,thick] (25bp,25bp) rectangle (75bp,75bp);
    \draw (50bp,0bp) node[below] {third application};
  \end{scope}

  \begin{scope}[xshift=10cm]
    \draw (50bp,50bp) node{\pgfuseshading{myshadingF}};
    \draw[white,thick] (50bp,50bp) circle [radius=25bp];
    \draw (50bp,0bp) node[below] {fourth application};
  \end{scope}
\end{tikzpicture}
```

An advantage of this approach is that when you rotate a radial shading, no distortion is introduced:



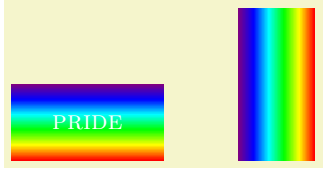
```
\pgfdeclareradialshading{ballshading}{\pgfpoint{-10bp}{10bp}}
 {color(0bp)=(red!15!white); color(9bp)=(red!75!white);
 color(18bp)=(red!70!black); color(25bp)=(red!50!black); color(50bp)=(black)}
\pgfuseshading{ballshading}
\hskip 1cm
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{1cm}{1cm}}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{3cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{0}
  \pgfusepath{}
  \pgfpathcircle{\pgfpoint{6cm}{0cm}}{1cm}
  \pgfshadepath{ballshading}{45}
  \pgfusepath{}
\end{pgfpicture}
```

If you specify a rotation of 90° and if the path is not a square, but an elongated rectangle, the "desired" effect results: The shading will exactly vary between the colors at the 25bp and 75bp boundaries. Here is an example:



```
\pgfdeclareverticalshading{myshadingG}{100bp}
 {color(0bp)=(red); color(25bp)=(green);  color(75bp)=(blue);  color(100bp)=(black)}
\begin{pgfpicture}
  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{0}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{3cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{90}
  \pgfusepath{stroke}
  \pgfpathrectangle{\pgfpoint{6cm}{0cm}}{\pgfpoint{2cm}{1cm}}
  \pgfshadepath{myshadingG}{45}
  \pgfusepath{stroke}
\end{pgfpicture}
```

As a final example, let us define a "rainbow spectrum" shading for use with Ti*k*Z.




```
\pgfdeclareverticalshading{rainbow}{100bp}
 {color(0bp)=(red); color(25bp)=(red); color(35bp)=(yellow);
  color(45bp)=(green); color(55bp)=(cyan); color(65bp)=(blue);
  color(75bp)=(violet); color(100bp)=(violet)}
\begin{tikzpicture}[shading=rainbow]
  \shade (0,0) rectangle node[white] {\textsc{pride}} (2,1);
  \shade[shading angle=90] (3,0) rectangle +(1,2);
\end{tikzpicture}
```

Note that rainbow shadings are *way* too colorful in almost all applications.

`\pgfsetadditionalshadetransform{`⟨*transformation*⟩`}`

This command allows you to specify an additional transformation that should be applied to shadings when the `\pgfshadepath` command is used. The ⟨*transformation*⟩ should be transformation code like `\pgftransformrotate{20}`.

115 Transparency

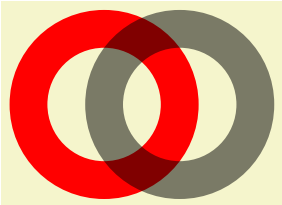
For an introduction to the notion of transparency, fadings, and transparency groups, please consult Section 23.

115.1 Specifying a Uniform Opacity

Specifying a stroke and/or fill opacity is quite easy.

\pgfsetstrokeopacity{⟨*value*⟩}

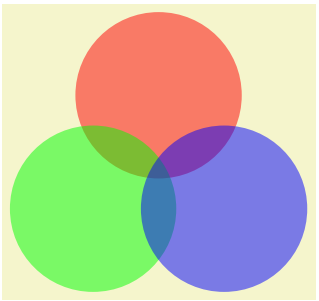
Sets the opacity of stroking operations. The ⟨*value*⟩ should be a number between `0` and `1`, where `1` means "fully opaque" and `0` means "fully transparent". A value like `0.5` will cause paths to be stroked in a semitransparent way.



```
\begin{pgfpicture}
  \pgfsetlinewidth{5mm}
  \color{red}
  \pgfpathcircle{\pgfpoint{0cm}{0cm}}{10mm} \pgfusepath{stroke}
  \color{black}
  \pgfsetstrokeopacity{0.5}
  \pgfpathcircle{\pgfpoint{1cm}{0cm}}{10mm} \pgfusepath{stroke}
\end{pgfpicture}
```

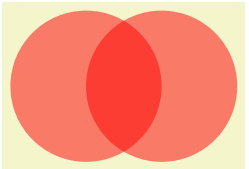
\pgfsetfillopacity{⟨*value*⟩}

Sets the opacity of filling operations. As for stroking, the ⟨*value*⟩ should be a number between `0` and `1`. The "filling transparency" will also be used for text and images.



```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red]   (90:1cm)  circle [radius=11mm];
  \fill[green] (210:1cm) circle [radius=11mm];
  \fill[blue]  (-30:1cm) circle [radius=11mm];
\end{tikzpicture}
```

Note the following effect: If you set up a certain opacity for stroking or filling and you stroke or fill the same area twice, the effect accumulates:



```
\begin{tikzpicture}
  \pgfsetfillopacity{0.5}
  \fill[red] (0,0) circle [radius=1];
  \fill[red] (1,0) circle [radius=1];
\end{tikzpicture}
```

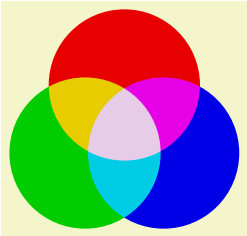
Often, this is exactly what you intend, but not always. You can use transparency groups, see the end of this section, to change this.

115.2 Specifying a Blend Mode

To set the blend mode, use the following command:

\pgfsetblendmode{⟨*mode*⟩}

Sets the blend mode to one of the values described in Section 23.3. As described there, blend modes are an advanced feature of PDF and not always rendered correctly.



```
\tikz [transparency group] {
  \pgfsetblendmode{screen}

  \fill[red!90!black]   ( 90:.6) circle [radius=1];
  \fill[green!80!black] (210:.6) circle [radius=1];
  \fill[blue!90!black]  (330:.6) circle [radius=1];
}
```

115.3 Specifying a Fading

The method used by PGF for specifying fadings is quite general: You "paint" the fading using any of the standard graphics commands. In more detail: You create a normal picture, which may even contain text, image, and shadings. Then, you create a fading based on this picture. For this, the *luminosity* of each pixel of the picture is analyzed (the brighter the pixel, the higher the luminosity – a black pixel has luminosity 0, a white pixel has luminosity 1, a gray pixel has some intermediate value as does a red pixel). Then, when the fading is used, the luminosity of the pixel determines the opacity of the fading at that position. Positions in the fading where the picture was black will be completely transparent, positions where the picture was white will be completely opaque. Positions that have not been painted at all in the picture are always completely transparent.

\pgfdeclarefading{⟨*name*⟩}{⟨*contents*⟩}

This command declares a fading named ⟨*name*⟩ for later use. The "picture" on which the fading is based is given by the ⟨*contents*⟩. The ⟨*contents*⟩ are normally typeset in a TEX box. The resulting box is then used as the "picture". In particular, inside the ⟨*contents*⟩ you must explicitly open a `{pgfpicture}` environment if you wish to use PGF commands.

Let's start with an easy example. Our first fading picture is just some text:



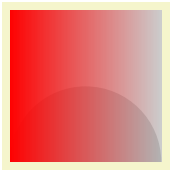

```
\pgfdeclarefading{fading1}{\textcolor{white}{Ti\emph{k}Z}}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading1}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

What's happening here? The "fading picture" is mostly transparent, except for the pixels that are part of the word Ti*k*Z. Now, these pixels are *white* and, thus, have a high luminosity. This in turn means that these pixels of the fading will be highly opaque. For this reason, only those pixels of the big red rectangle "shine through" that are at the positions of these opaque pixels.

It is somewhat counter-intuitive that the white pixels in a fading picture are opaque in a fading. For this reason, the color `pgftransparent` is defined to be the same as `black`. This allows one to write `pgftransparent` for completely transparent parts of a fading picture and `pgftransparent!0` for the opaque parts and things like `pgftransparent!20` for parts that are 20% transparent.

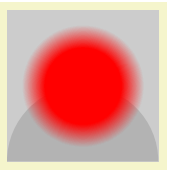
Furthermore, the color `pgftransparent!0` (which is the same as white and which corresponds to completely opaque) is installed at the beginning of a fading picture. Thus, in the above example the `\color{white}` was not really necessary.

Next, let us create a fading that gets more and more transparent as we go from left to right. For this, we put a shading inside the fading picture that has the color `pgftransparent!0` at the left-hand side and the color `pgftransparent!100` at the right-hand side.



```
\pgfdeclarefading{fading2}
{\tikz \shade[left color=pgftransparent!0,
              right color=pgftransparent!100] (0,0) rectangle (2,2);}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading2}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

In our final example, we create a fading that is based on a radial shading.



```
\pgfdeclareradialshading{myshading}{\pgfpointorigin}
{
  color(0mm)=(pgftransparent!0);
  color(5mm)=(pgftransparent!0);
  color(8mm)=(pgftransparent!100);
  color(15mm)=(pgftransparent!100)
}
\pgfdeclarefading{fading3}{\pgfuseshading{myshading}}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading3}{\pgftransformshift{\pgfpoint{1cm}{1cm}}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

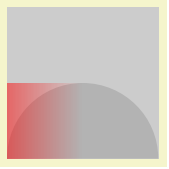
After having declared a fading, we can use it. As for shadings, there are different commands for using fadings:

\pgfsetfading{⟨*name*⟩}{⟨*transformations*⟩}

This command sets the graphic state parameter "fading" to a previously defined fading ⟨*name*⟩. This graphic state works like other graphic states, that is, is persists till the end of the current scope or until a different transparency setting is chosen.

When the fading is installed, it will be centered on the origin with its natural size. Anything outside the fading picture's original bounding box will be transparent and, thus, the fading effectively clips against this bounding box.

The ⟨*transformations*⟩ are applied to the fading before it is used. They contain normal PGF transformation commands like \pgftransformshift. You can also scale the fading using this command. Note, however, that the transformation needs to be inverted internally, which may result in inaccuracies and the following graphics may be slightly distorted if you use a strong ⟨*transformation*⟩.



```
\pgfdeclarefading{fading2}
{\tikz \shade[left color=pgftransparent!0,
              right color=pgftransparent!100] (0,0) rectangle (2,2);}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading2}{}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```



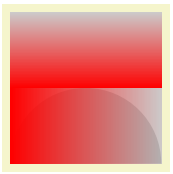
```
\pgfdeclarefading {fading2} {\tikz \shade [left color=pgftransparent!0, right
color=pgftransparent!100] (0,0) rectangle (2,2);}
\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);
  \pgfsetfading{fading2}{\pgftransformshift{\pgfpoint{1cm}{1cm}}
                         \pgftransformrotate{20}}
  \fill [red] (0,0) rectangle (2,2);
\end{tikzpicture}
```

\pgfsetfadingforcurrentpath{⟨*name*⟩}{⟨*transformations*⟩}

This command works like \pgfsetfading, but the fading is scaled and transformed according to the following rules:

1. If the current path is empty, the command has the same effect as \pgfsetfading.
2. Otherwise it is assumed that the fading has a size of 100bp times 100bp.
3. The fading is resized and shifted (using appropriate transformations) such that the position (25bp, 25bp) lies at the lower-left corner of the current path and the position (75bp, 75bp) lies at the upper-right corner of the current path.

Note that these rules are the same as the ones used in \pgfshadepath for shadings. After these transformations, the ⟨*transformations*⟩ are executed (typically a rotation).



```
\pgfdeclarehorizontalshading{shading}{100bp}
{ color(0pt)=(transparent!0);     color(25bp)=(transparent!0);
  color(75bp)=(transparent!100); color(100bp)=(transparent!100)}

\pgfdeclarefading{fading}{\pgfuseshading{shading}}

\begin{tikzpicture}
  \fill [black!20] (0,0) rectangle (2,2);
  \fill [black!30] (0,0) arc (180:0:1);

  \pgfpathrectangle{\pgfpointorigin}{\pgfpoint{2cm}{1cm}}
  \pgfsetfadingforcurrentpath{fading}{}
  \pgfusepath{discard}

  \fill [red] (0,0) rectangle (2,1);

  \pgfpathrectangle{\pgfpoint{0cm}{1cm}}{\pgfpoint{2cm}{1cm}}
  \pgfsetfadingforcurrentpath{fading}{\pgftransformrotate{90}}
  \pgfusepath{discard}

  \fill [red] (0,1) rectangle (2,2);
\end{tikzpicture}
```

\pgfsetfadingforcurrentpathstroked{⟨*name*⟩}{⟨*transformations*⟩}

This command works like \pgfsetfadingforcurrentpath, only the current path is enlarged by the line width in both x- and y-direction. This is exactly the enlargement necessary to compensate for the fact that if the current path will be stroked, this much needs to be added around the path's bounding box to actually contain the path.



```
\pgfdeclarehorizontalshading {shading}{100bp} { color(0pt)=(transparent!0);
color(25bp)=(transparent!0); color(75bp)=(transparent!100);
color(100bp)=(transparent!100)} \pgfdeclarefading {fading}{\pgfuseshading {shading}}
\begin{tikzpicture}
  \pgfsetlinewidth{2mm}
  \pgfpathmoveto{\pgfpointorigin}
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfsetfadingforcurrentpathstroked{fading}{}
  \pgfusepath{stroke}
\end{tikzpicture}
```

115.4 Transparency Groups

Transparency groups are declared using the following commands.

\begin{pgftransparencygroup}[⟨*options*⟩]
  ⟨*environment contents*⟩
\end{pgftransparencygroup}

This environment should only be used inside a {pgfpicture}. It has the following effect:

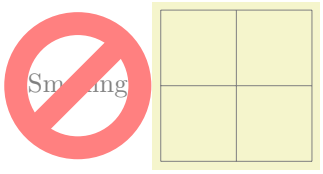
1. The ⟨*environment contents*⟩ are stroked/filled "ignoring any outside transparency". This means, all previous transparency settings are ignored (you can still set transparency inside the group, but never mind). This means that if in the ⟨*environment contents*⟩ you stroke a pixel three times in black, it is just black. Stroking it white afterwards yields a white pixel, and so on.
2. When the group is finished, it is painted as a whole. The *fill* transparency settings are now applied to the resulting picture. For instance, the pixel that has been painted three times in black and once in white is just white at the end, so this white color will be blended with whatever is "behind" the group on the page.

The optional ⟨*options*⟩ are keys that configure the transparency group further. Two keys are currently defined:

- knockout=⟨*true or false*⟩ Configures whether the group is a knockout group (if no argument is given, true is assumed; initially the key is always false, even when the command is used in a nested manner.) See Section 23.5 for details on knockout groups.

- `isolated=`⟨*true or false*⟩ Similar, but configures whether the group is an isolated group. Also see Section 23.5 for details on isolated groups.

Note that, depending on the driver, PGF may have to guess the size of the contents of the transparency group (because such a group is put in an XForm in PDF and a bounding box must be supplied). PGF will use normally use the size of the picture's bounding box at the end of the transparency group plus a safety margin of 1cm. Under normal circumstances, this will work nicely since the picture's bounding box contains everything anyway. However, if you have switched off the picture size tracking or if you are using canvas transformations, you may have to make sure that the bounding box is big enough. The trick is to locally create a picture that is "large enough" and then insert this picture into the main picture while ignoring the size. The following example shows how this is done:



```
\usetikzlibrary {shapes.symbols}
\begin{tikzpicture}
  \draw [help lines] (0,0) grid (2,2);

  % Stuff outside the picture, but still in a transparency group.
  \node [left,overlay] at (0,1) {
    \begin{tikzpicture}
      \pgfsetfillopacity{0.5}
      \pgftransparencygroup
      \node at (2,0) [forbidden sign,line width=2ex,draw=red,fill=white]
        {Smoking};
      \endpgftransparencygroup
    \end{tikzpicture}
  };
\end{tikzpicture}
```

`\pgftransparencygroup`
⟨*environment contents*⟩
`\endpgftransparencygroup`

The plain TEX version of the environment.

`\startpgftransparencygroup`
⟨*environment contents*⟩
`\stoppgftransparencygroup`

The ConTEXt version of the environment.

116 Animations

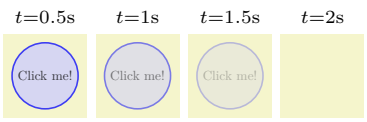
```
\usepgfmodule{animations}  % LaTeX and plain TeX when using pure pgf
\usepgfmodule[animations]  % ConTeXt when using pure pgf
```

This module contains the basic layer support of animations, which is documented in the following.

This section described the basic layer support of animations, the Ti*k*Z support is described in Section 26. As always, Ti*k*Z mainly converts syntactic constructs (like the special colon or quote syntax) to appropriate basic layer commands, which are documented here. Note, however, that while many attributes and options are the same on both layers, some things are handled differently on the basic layer.

116.1 Overview

An *animation* changes the way some part of a graphic looks like over time. The archetypical animation is, of course, a *movement* of node, but a change of, say, the opacity of a path is also an animation. PGF allows you to specify such animations using a set of commands and keys that are documented in the following.




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

Differently from other packages, the animations created by PGF are not precomputed sequences of pictures that are displayed in rapid succession. Rather, an animation created by PGF consists mainly of an annotation in the output that a certain attribute of a certain object should change over time in some specific way when the object is displayed. It is the job of the document viewer application to actually compute and display the animation. Interestingly, this means that animations neither increase the size of the output files noticeably nor does it put a special burden on TEX. The hard and complicated calculations are done by the viewer application, not by TEX and PGF.

Only few viewer applications and formats are currently "up to the job" of displaying animations. In particular, the popular PDF format does *not* allow one to specify animations in this way (one can partly "fake" animations at the high price of including a great number of precomputed pictures and using JavaScript in special viewers, but this is really not the same thing as what PGF does). Indeed, currently only the SVG format allows one to specify animations in a sensible way. Thus, PGF's animations will only be displayed when SVG is used as output format.

Because of the shortcomings of the other formats and, also, for purposes of printing and depicting animations in a sequential manner, PGF also allows you to create "snapshots" of animations. As an example, the following code shows how the same drawing is shown at different "time snapshots":




```
\usetikzlibrary {animations} \def \pgfname {\textsc {pgf}}
\tikz [make snapshot of=0.5s] \scoped :rotate = { 0s = "0", 2s = "90" }
  \node [draw=blue, very thick] {\pgfname};
\tikz [make snapshot of=1s]   \scoped :rotate = { 0s = "0", 2s = "90" }
  \node [draw=blue, very thick] {\pgfname};
\tikz [make snapshot of=1.5s] \scoped :rotate = { 0s = "0", 2s = "90" }
  \node [draw=blue, very thick] {\pgfname};
\tikz [make snapshot of=2s]   \scoped :rotate = { 0s = "0", 2s = "90" }
  \node [draw=blue, very thick] {\pgfname};
```

116.2 Animating an Attribute

116.2.1 The Main Command

Creating an animation is done using the command `\pgfanimateattribute`, which takes a to-be-animated attribute and options specifying the timeline:

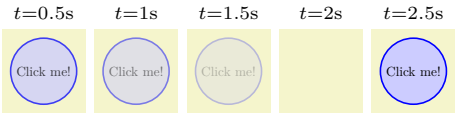
`\pgfanimateattribute{`⟨*attribute*⟩`}{`⟨*options*⟩`}`

Adds an animation of the ⟨*attribute*⟩ of a future *object* to the current graphic. Attributes are things like the "fill opacity" or the transformation matrix or the line width.

The ⟨*options*⟩ are keys that configure how the attribute changes over time. Using the `entry` key multiple times, you specify which value the chosen attribute should have at different points in time. Unless special keys are used, "outside" the specified timeline the animation has no effect:




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```



Other keys, like `repeats`, allow you to specify how the animation behaves "as a whole". These keys are documented later in this section.

The Attributes

In detail, the `\pgfanimateattribute` command opens a TEX-scope, looks up the *type* values of the specified ⟨*attribute*⟩ have (if you wish to animate the `opacity` of an object, the type is "scalar" meaning that entries must be scalar numbers; when you animate the `fill` attribute, the type is "color" and values must be colors, and so on), and then executes the ⟨*options*⟩ with the path prefix `/pgf/animation`. Finally, an appropriate system layer command `\pgfsysanimate...` is called to create the actual animation and the scope is closed.

The following ⟨*attributes*⟩ are permissible:

Attribute	*Type*
`draw`, `fill`	color
`line width`	dimension
`motion`	scalar
`opacity`, `fill opacity`, `draw opacity`	scalar
`path`	path
`rotate`	scalar
`scale`	scaling
`softpath`	softpath
`translate`	point
`view`	viewbox
`visible`	boolean
`stage`	boolean
`xskew`, `yskew`	scalar

These attributes are detailed in Sections 116.3 to 116.5, but here is a quick overview:

- `draw` and `fill` refer to the color used to draw (stroke) and fill paths in an object, respectively. Typical values for this attribute are `red` or `black!10`.
- `line width` is, of course, the line width used in an object. Typical values are `0.4pt` or `1mm`. Note that you (currently) cannot use keys like `thin` or `thick` here, but this may change in the future.
- `motion` is a slightly special attribute: It allows you to specify a path along which the object should be moved (using the `along` key). The values given to the `entry` key for this attribute refer to a *fraction of the distance along the path*. See the `along` key for details.
- `opacity` and the variants `fill opacity` and `draw opacity` animate the opacity of an object. Allowed values range between 0 and 1.
- `path` allows you to animate a path (it will morph). The "values" are now paths themselves. See Section 116.4 for details.
- `rotate` refers to a rotation of the object. Values for the `entry` key are the rotation angles like `0` or `90`.
- `scale` refers to the scaling of the object. Values are either single scalars values (like `1` or `1.5`) or two numbers separated by a comma (like `1,1.5` or `0.5,2`), referring to the x-scaling and y-scaling.

- `softpath` is a special case of the `path` attribute, see Section 116.4 once more.
- `translate` shifts the object by a certain vector. Values are points like `\pgfpoint{1cm}{2cm}`.
- `view` allows you to animate the view box of a view, see Section 116.5 for details.
- `visible` refers to the visibility of an object. Allowed values are `true` and `false`.
- `stage` is identical to `visible`, but when the object is not animated, it will be hidden by default.
- `xskew` and `yskew` skew the object. Attributes are angles like `0` or `45` or even `90`.

The Target Object

As stated earlier, the ⟨*options*⟩ are used to specify the object whose attribute for which an animation should be added to the picture. Indeed, you *must* specify the object explicitly using the `whom` key and you must do so *before* the object is created. Note that, in contrast, in SVG you can specify an animation more or less anywhere and then use hyper-references to link the animation to the to-be-animated object; PGF insists that you specify the animation before the object. This is a bit of a bother in some situations, but it is the only way to ensure that PGF has a fighting chance to attach some additional code to the object (which is necessary for almost all animations of the transformation matrix).

`/pgf/animation/whom=`⟨*id*⟩`.`⟨*type*⟩ (no default)

You *must* use this key once which each call of the `\pgfanimateattribute` command. The ⟨*id*⟩ and the optional ⟨*type*⟩ (which is whatever follows the first dot) will be passed to `\pgfidrefnextuse`, see that command for details.

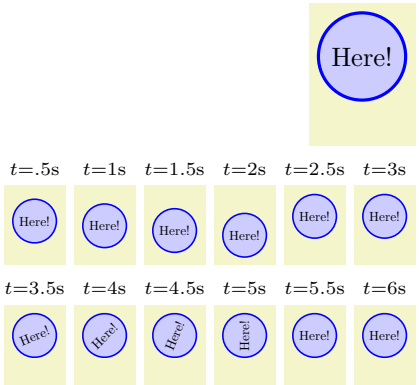
As explained in the introduction of this chapter, an "animation" is just a bit of special text in the output document asking a viewer application to animate the object at some later time. The `\pgfanimateattribute` command inserts this special text immediately, even though it refers to an object created only later on. Normally, this is not a problem, but the special text should be on the same page as the to-be-animated object. To ensure this, it suffices to call `\pgfanimateattribute` no earlier than the beginning of the `pgfpicture` containing the object.

Naming the Animation

You can assign a name to an animation for later (or early) reference. In particular, it is possible to begin *another* animation relative to the beginning or end of this animation and for referencing this animation must be assigned a name. See the `of` and `of next` keys for details.

`/pgf/animation/name=`⟨*name*⟩ (no default)

Assigns a name to the animation by which it can be referenced using the `of` and `of next` keys in another animation.




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {end, of next = my move animation, delay = 1s},
    entry = {0s}{0}, entry = {2s}{90}, begin snapshot = 3s, }
  \pgfanimateattribute{translate}{
    name = my move animation, whom = node, begin on = {click},
    entry = {0s}{\pgfpointorigin}, entry = {2s}{\pgfpoint{0cm}{-5mm}} }
  \node (node) [fill = blue!20, draw = blue, circle] {Here!};
}
```

`\pgfanimateattributecode{`⟨*attribute*⟩`}{`⟨*code*⟩`}`

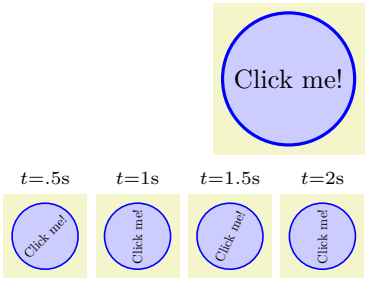
The command works like `\pgfanimateattribute`, only instead of ⟨*options*⟩ you specify some ⟨*code*⟩ whose job is to setup the options.

116.2.2 Specifying the Timeline

The core key for specifying how an attribute varies over time is the `entry` key:

/pgf/animation/entry={⟨*time*⟩}{⟨*value*⟩} (no default)

You use this key repeatedly to specify the different values that the ⟨*attribute*⟩ should have over time. At the ⟨*time*⟩ specified, the ⟨*attribute*⟩ will have the value specified as ⟨*value*⟩:




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click},
    entry = {0s}{0}, entry = {1s}{90}, entry = {1.1s}{45}, entry = {2s}{90}
  }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

You need to call `entry` once for each time in the timeline for which you want to specify a ⟨*value*⟩ explicitly. Between these times, the values get interpolated (see below for details). You need to specify the ⟨*time*⟩s in non-decreasing order (it is permissible and sometimes also necessary to specify the same time twice, namely to create a "jump" of the value of some attribute).

The ⟨*time*⟩ is parsed using the command `\pgfparsetime` described later.

Start and end of the timeline. The first and last times of the timeline are a bit special: The timeline starts on the first time and the duration of the timeline is the difference between the first and last time. "Starting" on the start time actually means that any beginnings (see the `begin` and `end` keys) get offset by the start time; similarly end times are offset by this value.

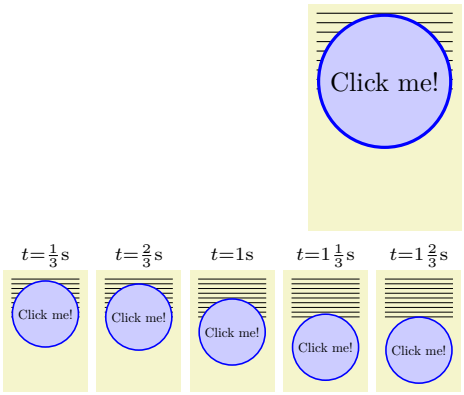
Syntax of the values. The syntax of the ⟨*value*⟩ varies according to the type of the ⟨*attribute*⟩. In detail, these are:

Type	*Syntax*
color	Standard color syntax like `red` or `black!10`
scalar	A value parsed using `\pgfmathparse`
dimension	A dimension parsed using `\pgfmathparse`
path	A sequence of path construction commands
softpath	A sequence of soft path construction commands
scaling	A scalar value or a pair of scalar values separated by a comma
point	A PGF-point like `\pgfpoint{1cm}{5mm}`
viewbox	Two PGF-points
boolean	`true` or `false`

Interpolation between key times. You use the `entry` key repeatedly, namely once for each "key time", which is a time point for which you specify the value of the attribute explicitly. Between these key times, the attribute's value is interpolated. Normally, this is just a linear interpolation, but you can influence this using the following keys, see Section 26.5.4 for details.

/pgf/animations/exit control={⟨*time fraction*⟩}{⟨*value fraction*⟩} (no default)

Same as `/tikz/animate/options/exit control`.




```
\usepgfmodule {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \pgfanimateattribute{translate}{
    whom = node, begin on = {click},
    exit control={1}{0},
    entry = {0s}{\pgfpointorigin},
    linear, % revert to default
    entry = {1s}{\pgfpoint{0cm}{-5mm}},
    entry control={0}{1},
    entry = {2s}{\pgfpoint{0cm}{-10mm}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

/pgf/animations/entry control={⟨*time fraction*⟩}{⟨*value fraction*⟩} (no default)

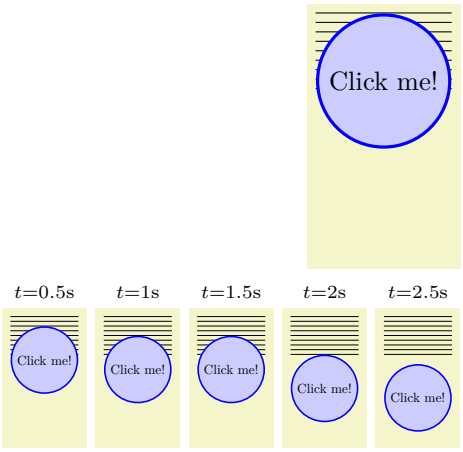
Works like `exit control`.

`/pgf/animations/linear` (no value)

A shorthand for `exit control={0}{0}, entry control={1}{1}`. This will (re)install a linear curve.

`/pgf/animations/stay` (no value)

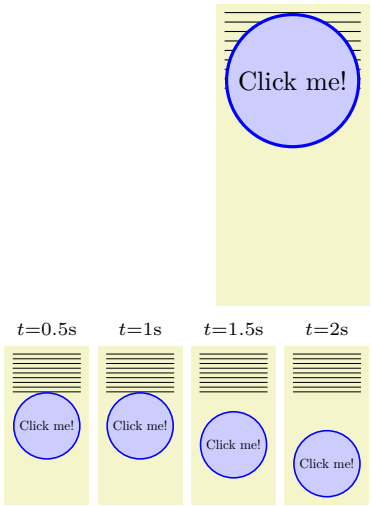
Same as `/tikz/animate/options/stay`.




```
\usepgfmodule {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \pgfanimateattribute{translate}{
    whom = node, begin on = {click},
    entry = {0s}{\pgfpointorigin},
    stay,
    entry = {1s}{\pgfpoint{0cm}{-5mm}},
    linear,
    entry = {2s}{\pgfpoint{0cm}{-10mm}},
    entry = {3s}{\pgfpoint{0cm}{-15mm}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`/pgf/animations/jump` (no value)

Same as `/tikz/animate/options/jump`.

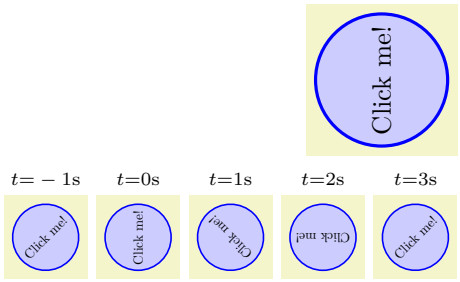



```
\usepgfmodule {animations}
\tikz {
  \foreach \i in {0,0.125,...,1} \draw (-0.9,.9-\i) -- ++(1.8,0);
  \pgfanimateattribute{translate}{
    whom = node, begin on = {click},
    entry = {0s}{\pgfpointorigin},
    jump,
    entry = {1s}{\pgfpoint{0cm}{-1cm}},
    linear,
    entry = {2s}{\pgfpoint{0cm}{-2cm}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

When the time of an animation lies outside the timeline specified by the `entry` keys, no animation is present. This means that the value of the attribute is the object's scope is used instead. Using the following key, you can set this value directly:

`/tikz/animations/base=`⟨*value*⟩ (no default)

The syntax of the ⟨*value*⟩ is the same as for the `entry` key. The ⟨*value*⟩ is installed as the value of the object's attribute whenever the timeline is not active. This makes it easy to specify the value of an attribute when the animation is "not running".




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click},
    entry = {0s}{90}, entry = {2s}{180},
    base  = 45
  }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

It may happen that there is more than one timeline active that is "trying to modify" a given attribute. In this case, the following rules are used to determine, which timeline "wins":

1. If no animation is active at the current time (all animation either have not yet started or they have already ended), then the base value given in the animation encountered last in the code is used. (If there are no base values, the attribute is taken from the surrounding scope.)
2. If there are several active animations, the one that has started last is used and the its value is used.
3. If there are several active animations that have started at the same time, the one that comes last in the code is used.

Note that these rules do not apply to transformations of the canvas since these are always additive (or, phrased differently, they are always all active and the effects accumulate).

\pgfparsetime{⟨*time*⟩}

This command works like `\pgfmathparse` (indeed, it calls is internally), but returns the result in the macro `\pgftimeresult` rather than `\pgfmathresult`. Furthermore, the following changes are installed:

- The postfix operator `s` is added, which has no effect.
- The postfix operator `ms` is added, which divides a number by 1000, so `2ms` equals 0.002s.
- The postfix operator `min` is added, which multiplies a number by 60.
- The postfix operator `h` is added, which multiplies a number by 3600.
- The infix operator `:` is redefined, so that it multiplies its first argument by 60 and adds the second. This implies that `1:20` equals 80s and `01:00:00` equals 3600s.
- The parsing of octal numbers is switched off to allow things like `01:08` for 68s.

116.2.3 "Anti-Animations": Snapshots

There are a number of situations in which you want the "opposite" of an animation to happen: You want to create a "still image". For instance, when you want to print an animation you will typically wish to show one or more "temporal snapshots" of the animation. Also, you may wish to specify a value for an object when it is *not* being animated.

Let us start with creating a snapshot:

\pgfsnapshot{⟨*time*⟩}

When this command is used inside a TEX scope, the behavior of `\pgfanimateattribute` changes: Instead of adding an animation to the object and the attribute, the object's attribute is set to value it would have during the animation at time ⟨*time*⟩. Note that when this command is used in a TEX scope, no animation is created and no support by the driver is needed (so, it works with PDF).




```
\usetikzlibrary {animations}
\tikz [make snapshot of=1s,
       animate = { myself: = {
         :rotate = { 0s = "0", 2s = "90" },
         :color  = { 0s = "red", 2s = "green" },
         :line width = { 0s = "0mm", 4s = "4mm" }
       }}]
  \node [fill=black!20, draw] { Node };
```

Timing and Events. The timeline of an animation normally starts at a "moment `0s`" and the ⟨*time*⟩ is considered relative to this time. For instance, if a timeline contains, say, the settings `entry={2s}{0}` and `entry={3s}{10}` and `{`⟨*time*⟩`}` is set to `2.5s`, then the value the attribute will get is 5.

It is, however, also possible to specify that animations begin and end at certain times relative to events like a `click` event. *These events are not relevant with respect to snapshots.* However, there is one key that allows you to specify the beginning of the snapshot timeline:

`/tikz/animations/begin snapshot=`⟨*begin time*⟩ (no default)

When this key is used inside the options of `\pgfanimateattribute`, with respect to snapshots, the timeline begins at ⟨*begin time*⟩. This means that, if the snapshot time is set to ⟨*time*⟩ and the beginning of the snapshot's timeline is set to ⟨*begin time*⟩, the attribute is set to the value of the timeline at time ⟨*time*⟩ − ⟨*begin time*⟩.

The idea is that when you make a snapshot of several animations and all of them have started at different times because of different events, you use `begin snapshot` with each object and attribute to directly specify when these different events have happened.

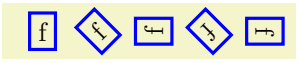
Note that the `end` keys have no effect with snapshots, that is, with a snapshot all animations always run till the end of the timeline (which may or may not be "forever").

Limitations. For snapshots, the value an animation has at time ⟨*time*⟩ must be computed by TEX. While in many cases this is easy to achieve, in some cases this is not trivial such as a timeline for a path with repeats plus smoothing via splines. An additional complication is the fact that an animation may be specified at a place far removed from the actual to-be-animated object. For these reasons, certain limitations apply to snapshots:

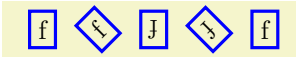
- The `begin` and `begin on` keys have no effect (but `begin snapshot` has one.
- The `end` and `end on` keys have no effect.
- The `current value` may not be used in a timeline (since PGF cannot really determine this value).
- The `accumulating` specification may not be used with paths, views, or motions.
- Since the timing computations are done using TEX code, they are not necessarily stable. For instance, when a time interval is very small and there are many repeats or when a spline is very complicated, the calculated values may not be fully accurate.

`\pgfsnapshotafter{`⟨*time*⟩`}`

This command works exactly like `\pgfsnapshot` only the "moment" that ⟨*time*⟩ refers to is conceptually ⟨*time*⟩ + ϵ: When timeline specifies several values for ⟨*time*⟩, this command will select the last value at ⟨*time*⟩, while `\pgfsnapshot` will select the first value at ⟨*time*⟩. Similarly, when a timeline ends at ⟨*time*⟩, `\pgfsnapshot` will select the last value of the timeline while `\pgfsnapshotafter` will not apply the animation any more:



```
\usepgfmodule {animations}
\foreach \t in {0,1,2,3,4} {
  \pgfsnapshot{\t}
  \tikz :rotate = { 0s = "0", 2s = "90", 2s = "180", 4s = "270" }
    \node [draw=blue, very thick] {f}; }
```

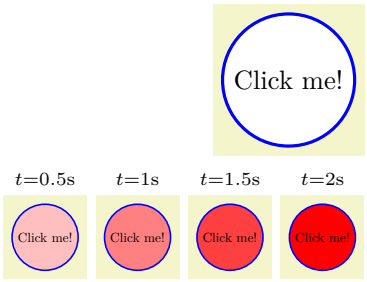


```
\usepgfmodule {animations}
\foreach \t in {0,1,2,3,4} {
  \pgfsnapshotafter{\t}
  \tikz :rotate = { 0s = "0", 2s = "90", 2s = "180", 4s = "270" }
    \node [draw=blue, very thick] {f}; }
```

116.3 Animating Color, Opacity, Visibility, and Staging

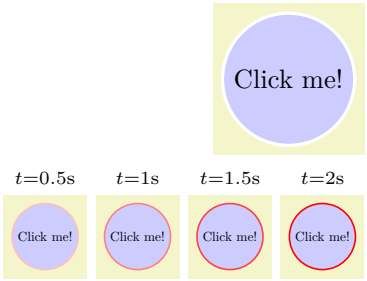
`\pgfanimateattribute{fill}{`⟨*options*⟩`}`

You can animate the color of the target object of an animation using the attributes `fill` or `draw`, which animate the fill color and the drawing (stroking) color, respectively. To animate both the fill and draw color, you need to create two animations, one for each.



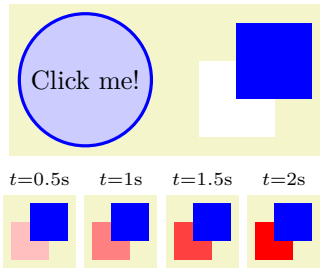

```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill}{
    whom = node.background, begin on = {click},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`\pgfanimateattribute{draw}{`⟨*options*⟩`}`




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{draw}{
    whom = node.background, begin on = {click},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

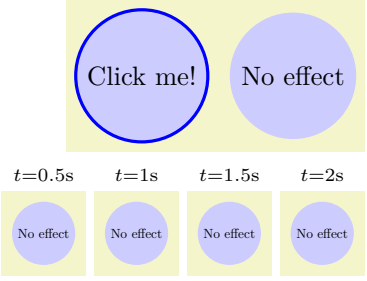
When the target of a color animation is a scope, you animate the color "used in this scope" for filling or stroking. However, when an object inside the scope has its color set explicitly, this color overrules the color of the scope:



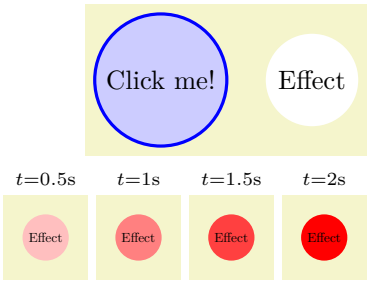

```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill}{
    whom = example, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \begin{scope}[name = example]
    \fill (1.5,-0.75) rectangle ++ (1,1);
    \fill [blue] (2,-0.25) rectangle ++ (1,1);
  \end{scope}
}
```

Note that in certain cases, a graphic scope may contain graphic objects with their colors set explicitly "in places where you do not expect it": In particular, a node normally consists at least of a background path and a text. For both the text and for the background path, colors will be set for the text and also for the path explicitly. This means that when you pick the fill attribute of a node as the target of an animation, you will *not* animate the color of the background path in case this color has been set explicitly. Instead, you must choose the background path of the node as the target of the animation. Fortunately, this is easy to achieve since when the background path of a node is created, the identifier type is set to `background`, which in turn allows you to access it as ⟨*node*⟩`.background` through the `whom` key.

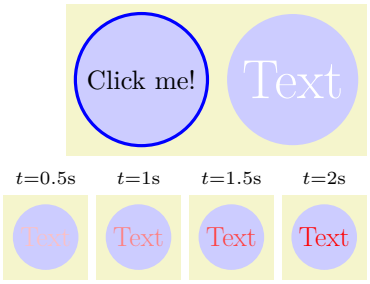
The text of a node also gets it color set explicitly, which means that a change of the node's scope's color has no effect on the text color. Instead, you must choose ⟨*name*⟩`.text` as the target (or, if the node has more parts, use the name of the part as the identifier type instead of `text`).




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill}{
    whom = example, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (2,0) (example) [fill = blue!20, circle] {No effect}; }
```



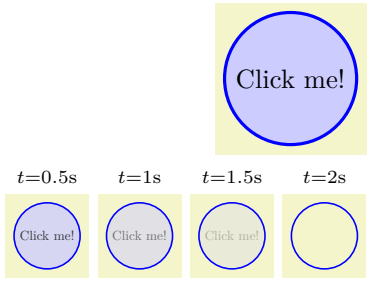

```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill}{
    whom = example.background, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (2,0) (example) [fill = blue!20, circle] {Effect}; }
```




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill}{
    whom = example.text, begin on = {click, of next=node},
    entry = {0s}{white}, entry = {2s}{red} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (2,0) (example) [fill = blue!20, circle, font=\huge] {Text}; }
```

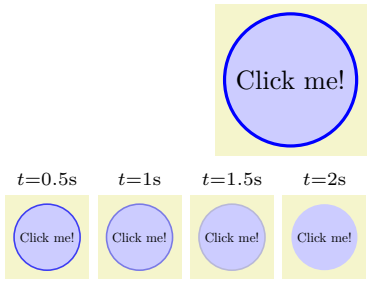
Similarly to the color, you can also set the opacity used for filling and for drawing. You specify the opacity using a number between 0 (transparent) and 1 (opaque).

`\pgfanimateattribute{fill opacity}{`*⟨options⟩*`}`




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{fill opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

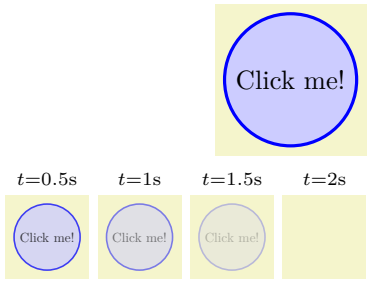
`\pgfanimateattribute{draw opacity}{`*⟨options⟩*`}`




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{draw opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`\pgfanimateattribute{opacity}{`*⟨options⟩*`}`

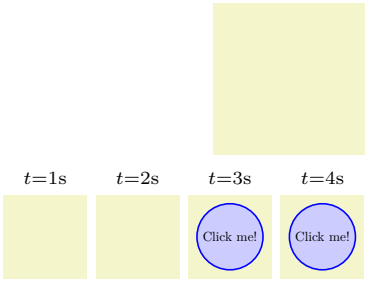
Unlike colors, where there is no joint attribute for filling and stroking, there is a single `opacity` attribute in addition to the above two attributes. If supported by the driver, it treats the graphic object to which it is applied as a transparency group. In essence, "this attribute does what you want" at least in most situations.




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{opacity}{
    whom = node, begin on = {click}, entry = {0s}{1}, entry = {2s}{0} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`\pgfanimateattribute{visible}{`*options*`}`

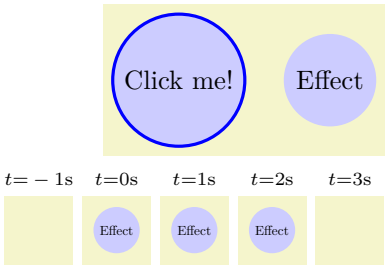
The difference between the `visible` attribute and an opacity of `0` is that an invisible object cannot be clicked and does not need to be rendered. The (only) two possible values for this attribute are `false` and `true`.




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{visible}{
    whom = node, begin on = {click}, entry = {0s}{false}, entry = {2s}{false} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

`\pgfanimateattribute{stage}{`*options*`}`

This attribute is the same as the `visible` attribute, only `base=false` is set by default. This means that the object is *only* visible when you explicitly during the time the entries are set to `true`. The idea behind the name "stage" is that the object is normally "off stage" and when you explicitly set the "stage attribute" to `true` the object "enters" the stage and "leaves" once more when it is no longer "on stage".

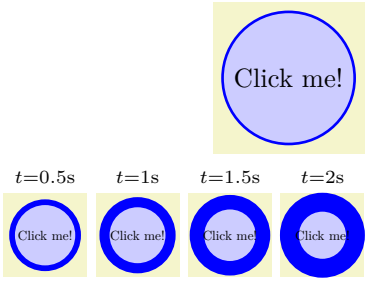



```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{stage}{
    whom = example, begin on = {click, of next=node},
    entry = {0s}{true}, entry = {2s}{true} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \node at (2,0) (example) [fill = blue!20, circle] {Effect}; }
```

116.4 Animating Paths and their Rendering

You can animate the appearance of a path in the following ways:

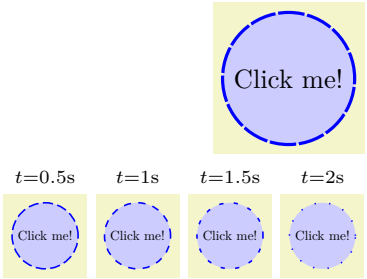
`\pgfanimateattribute{line width}{`*options*`}`



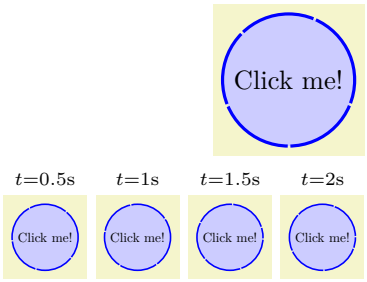

```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{line width}{
    whom = node, begin on = {click}, entry = {0s}{1pt}, entry = {2s}{5mm} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```

The possible values passed to the `entry` key are, of course, dimensions.

`\pgfanimateattribute{dash}{`*options*`}`




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{dash}{
    whom = node, begin on = {click}, entry = {0s}{{{10pt}{1pt}}{0pt}},
                                     entry = {2s}{{{1pt}{10pt}}{0pt}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

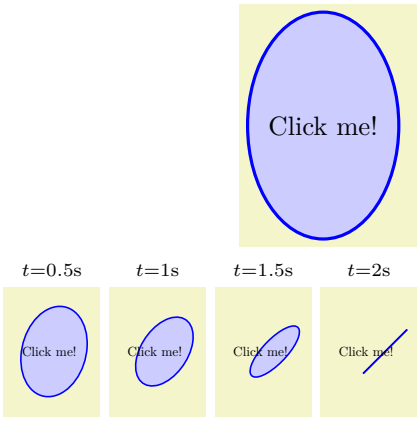



```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{dash}{
    whom = node, begin on = {click}, entry = {0s}{{{1cm}{1pt}}{0pt}},
                                     entry = {2s}{{{1cm}{1pt}}{1cm}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

To specify the dash pattern, you specify a sequence of "on and off" dimensions; see `\pgfsetdash` for details. Note that you *must* specify the same number of elements in all patterns of a timeline: You cannot specify that the dash pattern for `1s` is `{1pt}{2pt}` and for `2s` is `{1pt}{3pt}{2pt}` since the number of elements would differ. In particular, you cannot (sensibly) use `current value` for the first entry since this corresponds to an empty dash pattern (even when you have specified a dash pattern for the target object: this pattern will not be attached to the to-be-animated scope or object but to a surrounding scope and, thus, the to-be-animated scope will not have any dash pattern installed).

`\pgfanimateattribute{path}{`⟨*options*⟩`}`

You can animate the path itself:




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{path}{
    whom = node.background.path, begin on = {click, of next=node},
    entry = {0s}{\pgfpathellipse{\pgfpointorigin}
                  {\pgfpointxy{1}{0}}{\pgfpointxy{0}{1.5}}},
    entry = {2s}{\pgfpathellipse{\pgfpointxy{.5}{0}}
                  {\pgfpointxy{.5}{.5}}{\pgfpointxy{0.25}{.25}}}}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
}
```

The path is specified by giving path construction commands as in the above example. They will be executed in a special protected scope to ensure that they have only little side effects.

As for the dash pattern, you must ensure that all paths in the timeline have the same structure (same sequence of path construction commands); only the coordinates may differ. In particular, you cannot say that the path at `1s` is a rectangle using `\pgfpathrectangle` and at `2s` is a circle using `\pgfpathcircle`. Instead, you would have to ensure that at both times that path consists of appropriate Bézier curves.

Unlike the dash pattern, the to-be-animated object is, indeed, the path itself and not some special scope. This means that you can use the `current value` for the start path. However, this also means that you really must pick *the path object* as the target of the animation. In conjunction with Ti*k*Z, this will be an object of type `path` as in the above example.

When a path is animated, it cannot have "normal" arrows attached to it since due to the way PGF adds arrow tips to paths, these would not "move along" with the path (you get an error message if you try). However, it still *is* possible to add arrow tips to an animated path, only you need to use the `arrows` key described next.

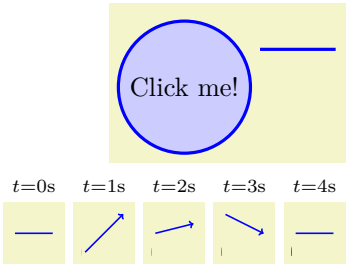
Concerning the bounding box computation of the path, a bounding box for all paths mentioned for any time point is used as the overall bounding box.

`/pgf/animation/arrows=`⟨*start tip spec*⟩`-`⟨*end tip spec*⟩ (no default)

This key specifies arrow tips during the animation of the path. The syntax for the arrow tips is the same syntax as the `\pgfsetarrow` command or Ti*k*Z's `arrows` key. The specified start and end arrow tips are rendered as "markers", which are added to the path *only* during the animation. The markers are rotated along with the path in exactly the same way as normal arrow tips would be. To be precise, the rules used for the computation of where arrow tips go and in which direction they head is not always the same for "static" arrow tips (arrow tips added to a normal path) and the "dynamic" arrow tips based

on markers; namely when the paths are very short or closed. For this reason, you should add arrow tips to animated paths only when the paths are "nice and simple" in the sense that they consist of a single segment whose ends are reasonably long.

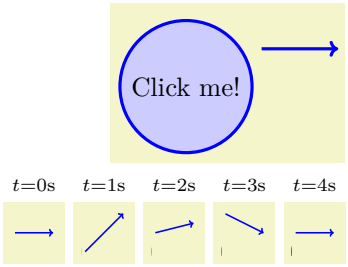
In addition to adding the arrow tips to the path during the animation, the path gets shortened as necessary to compensate for the extend of the arrow tips. However, for this to work, the arrow tips have to be specified before path values are specified (since the shortening is done immediately when a path value is parsed).




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{path}{
    whom = p.path, begin on = {click, of next=node}, arrows = ->,
    entry = {1s}{\pgfpathmoveto{\pgfpoint{1cm}{0cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{1cm}}},
    entry = {3s}{\pgfpathmoveto{\pgfpoint{1cm}{1cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{5mm}}}}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \draw [very thick, blue, name=p] (1,0.5) -- (2,0.5);
}
```

Note that the markers that visualize the arrow tips are rendered only once per animation. In consequence, "bending" arrow tips cannot be rendered correctly: As a path "morphs" a bend arrow tip needs not only to rotate along, but must actually "bend along", which is not supported (neither by PGF nor by SVG).

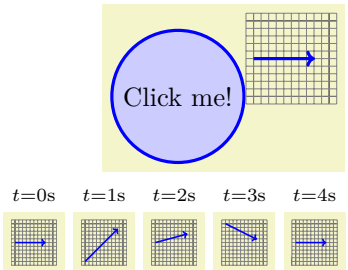
As pointed out earlier, an animated path cannot have "static" arrow tips. However, when you specify a `base` value, which is the path used whenever there is no active animation, *will* use the arrow tips. As a result, you can use this to animate a path with an arrow tip:




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{path}{
    whom = p.path, begin on = {click, of next=node}, arrows = ->,
    base = {\pgfpathmoveto{\pgfpoint{1cm}{5mm}}
            \pgfpathlineto{\pgfpoint{2cm}{5mm}}},
    entry = {1s}{\pgfpathmoveto{\pgfpoint{1cm}{0cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{1cm}}},
    entry = {3s}{\pgfpathmoveto{\pgfpoint{1cm}{1cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{5mm}}}}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \draw [very thick, blue, name=p];
}
```

`/pgf/animation/shorten >`=⟨*dimension*⟩ (no default)

Just like the normal TikZ key `shorten >`, this key specifies an extra shortening of to-be-animated paths. Whenever a path is parsed as a value for a path animation, it gets shortened at the end by the ⟨*dimension*⟩ (and, additionally, by the length of the attached arrow tip). Just like the `arrows` key, this key must be given before the path entries are specified.




```
\usepgfmodule {animations}
\tikz {
  \pgfanimateattribute{path}{
    whom = p.path, begin on = {click, of next=node}, arrows = ->,
    shorten > = 2mm,
    base = {\pgfpathmoveto{\pgfpoint{1cm}{5mm}}
            \pgfpathlineto{\pgfpoint{2cm}{5mm}}},
    entry = {1s}{\pgfpathmoveto{\pgfpoint{1cm}{0cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{1cm}}},
    entry = {3s}{\pgfpathmoveto{\pgfpoint{1cm}{1cm}}
                 \pgfpathlineto{\pgfpoint{2cm}{5mm}}}}
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};
  \draw               (0.9,-0.1) grid (2.1,1.1);
  \draw [help lines] (0.9,-0.1) grid[step=1mm] (2.1,1.1);
  \draw [very thick, blue, name=p];
}
```

`/pgf/animation/shorten <`=⟨*dimension*⟩ (no default)

Works like `shorten >`.

116.5 Animating Transformations and Views

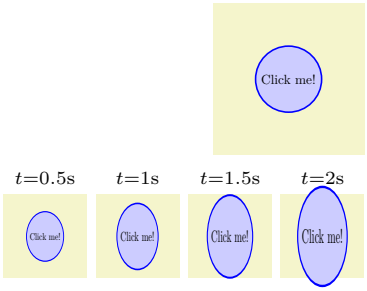
In order to animate the canvas transformation matrix, you do not animate an attribute called "`transform`" (or something similar). Rather, there are several keys that all manipulate the canvas transformation matrix in different ways. These keys, taken in appropriate combination, allow you to achieve any particular canvas transformation matrix. All keys that animate the transformation matrix *always* accumulate.

Some, but not all, of these keys also have an effect on the bounding box computation: The `translate` and `motion` attribute change the computation of the bounding box in such a way that it is chosen large enough as to include the whole picture during all stages of the animation (however, if there are multiple transformations active at the same time, the computation may not be correct). In contrast, `scale`, `rotate` and `skew` animations change the canvas transformation, but are ignored for the bounding box computation. When in doubt, please set the bounding box explicitly.

Let us start with the basic keys that allow you to change the canvas transformation matrix directly:

`\pgfanimateattribute{scale}{`⟨*options*⟩`}`

The `scale` attribute adds an animation of the scaling:

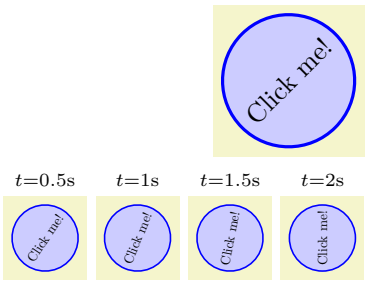



```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{scale}{
    whom = node, begin on = {click},
    entry = {0s}{0.5}, entry = {2s}{0.75,1.5} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```

The values passed to the `entry` key must either be single scalar values or a pair of such numbers separated by a comma (which then refer to the x- and y-scaling).

`\pgfanimateattribute{rotate}{`⟨*options*⟩`}`

The `rotate` key adds an animation of the rotation:

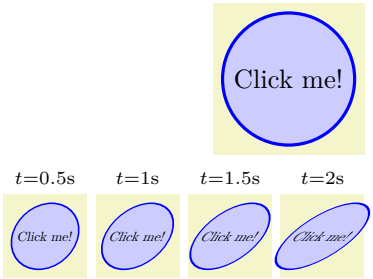



```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click},
    entry = {0s}{45}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```

The values are scalar values representing a rotation in degrees.

`\pgfanimateattribute{xskew}{`⟨*options*⟩`}`

The `xskew` and `yskew` keys (and also `skew x` and `skew y`, which are aliases) add an animation of the skew (given in degrees, not as a slant):




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{xskew}{
    whom = node, begin on = {click}, entry = {0s}{0}, entry = {2s}{45} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```

The values are scalar values.

`\pgfanimateattribute{yskew}{`⟨*options*⟩`}`

See `xskew`.

`\pgfanimateattribute{skew x}{⟨options⟩}`

An alias of `xskew`.

`\pgfanimateattribute{skew y}{⟨options⟩}`

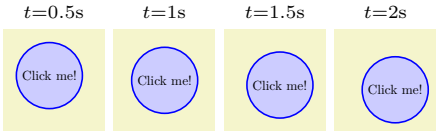
An alias of `yskew`.

`\pgfanimateattribute{translate}{⟨options⟩}`

The `translate` key adds an animation of the translation (shift):




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{translate}{
    whom = node, begin on = {click},
    entry = {0s}{\pgfpointorigin}, entry = {2s}{\pgfpoint{5mm}{-5mm}} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```



The values are PGF-points.

Unlike for the previous canvas transformations, for this key the bounding box computation is changed: All points in the to-be-animated scope do not only contribute to the normal bounding box, but they also contribute shifted by all points in the entry list. The effect is that a bounding box is computed that encompasses the animated scope at all stages.

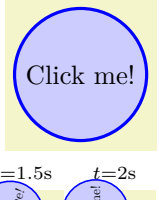
For all of these attributes, the following key is of importance:

`/pgf/animation/origin=`⟨*pgf point*⟩ (no default)

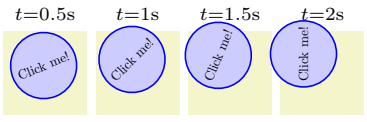
An animation of the canvas transformation is added to all other transformations from surrounding or interior scopes. This means that, in particular, the origin of a canvas transformation is, by default, the origin of the canvas of the scope surrounding the transformation object.

For some canvas animations, like a rotation or a scaling, you will typically wish to use a different origin (like the center of an object that is to be rotated or scaled). You can achieve this effect by surrounding the object by a scope that shifts the canvas to the desired origin, followed by a scope whose transformation matrix you animate, followed by a scope that shifts back the canvas.

The `origin` key simplifies this process by allowing you to specify the origin of the transformation directly. Internally, however, all this key does is to create the above-mentioned scopes with the necessary shifts.

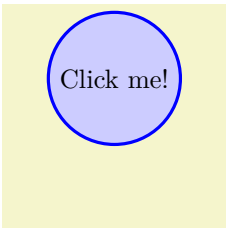



```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click},
    origin = \pgfpoint{-5mm}{0mm}, entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!};
}
```

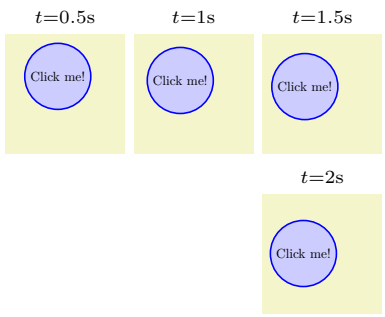


`\pgfanimateattribute{motion}{⟨options⟩}`

A second way of changing the canvas transformation matrix is to use the `motion` attribute:




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{motion}{
    whom = node, begin on = {click},
    along = \pgfpathcircle{\pgfpointorigin}{5mm},
    entry = {0s}{.25}, entry = {2s}{.5} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```



Just like the `translate` attribute, this key also changes the bounding box computation.

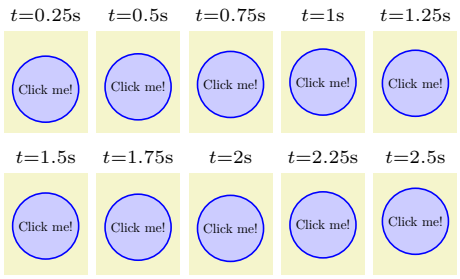
`/pgf/animation/along`=⟨*path*⟩ (no default)

This key must be used with `motion` attribute to specify a path along which the transformation matrix will be "moved" (that is, a shift transformation will be added to the different points on the path).

The values passed to the `entry` key specify fractions of the distance along the ⟨*path*⟩. That means, when you provide a value of `0`, you reference the start point of the path, a value of `1` references the end of the path and `0.5` referenced the point halfway along the path.




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{motion}{
    whom = node, begin on = {click},
    along = \pgfpathmoveto{\pgfpointorigin}
            \pgfpathlineto{\pgfpoint{0mm}{5mm}},
    entry = {0s}{0}, entry = {1s}{0.5}, entry = {2s}{0.25}, entry={3s}{1} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```

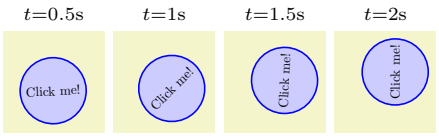


`/pgf/animation/rotate along`=⟨*Boolean*⟩ (default `true`)

When set to `true`, the `along` key additionally adds a rotation that varies in such a way that a tangent to the path always points right.




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{motion}{
    whom = node, begin on = {click},
    rotate along = true,
    along = \pgfpathmoveto {\pgfpointorigin}
            \pgfpathcurveto{\pgfpoint{5mm}{0cm}}{\pgfpoint{5mm}{0cm}}
                           {\pgfpoint{5mm}{5mm}},
    entry = {0s}{0}, entry = {2s}{1} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```

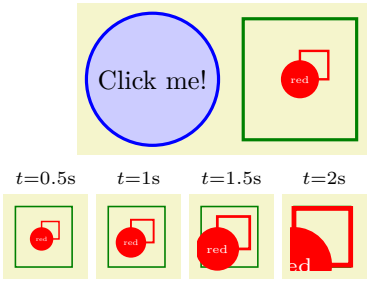


The final method of changing the transformation matrix is to animate a *view*.

`\pgfanimateattribute{view}{⟨options⟩}`

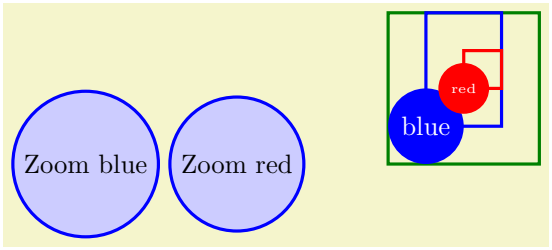
A view is a canvas transformation that shifts and scales the canvas in such a way that a certain rectangle "matches" another rectangle: The idea is that you "look through" a "window" (the view) and "see" a certain area of the canvas. View animation do not allow you to do anything that cannot also be done using the `translate` and `scale` keys in combination, but it often much more natural to animate which area of a graphic you wish to see than to compute and animate a scaling and shift explicitly.

In order to use a view, you first need to create a view, which is done using a `{pgfviewboxscope}`, see Section 108.3.2, which is used by the `views` library internally. You can then animate the view using the `view` attribute. The values passed to the `entry` key must be two PGF-points, each surrounded by parentheses.




```
\usepgfmodule {animations} \usetikzlibrary {views}
\tikz [very thick] {
  \pgfanimateattribute{view}{
    whom = me.view, begin on = {click, of next=node}, freeze at end,
    entry = {0s}{{\pgfpoint{0mm}{0mm}}{\pgfpoint{20mm}{20mm}}},
    entry = {2s}{{\pgfpoint{10mm}{10mm}}{\pgfpoint{15mm}{15mm}}} }
  \node (node) [fill = blue!20, draw = blue, very thick, circle] {Click me!};

  \draw [green!50!black] (1.2,-0.8) rectangle (2.7,0.8);
  \begin{scope}[name = me, view = {(0,0) (2,2) at (1.2,-0.8) (2.7,0.8)}]
    \draw [red] (10mm,10mm) rectangle (15mm,15mm);
    \node at (10mm,10mm) [circle, fill=red, text=white, font=\tiny] {red};
  \end{scope}
}
```




```
\usepgfmodule {animations} \usetikzlibrary {views}
\tikz [very thick] {
  \pgfanimateattribute{view}{
    whom = me.view, begin on = {click, of next=n1}, freeze at end,
    entry = {0s}{\pgfpoint{0mm}{0mm}}{\pgfpoint{2cm}{2cm}},
    entry = {2s}{{\pgfpoint{5mm}{5mm}}{\pgfpoint{15mm}{20mm}}} }
  \pgfanimateattribute{view}{
    whom = me.view, begin on = {click, of next=n2}, freeze at end,
    entry = {0s}{\pgfpoint{0mm}{0mm}}{\pgfpoint{2cm}{2cm}},
    entry = {2s}{{\pgfpoint{10mm}{10mm}}{\pgfpoint{15mm}{15mm}}} }
  \node (n1) at (0,0) [fill = blue!20, draw = blue, circle] {Zoom blue};
  \node (n2) at (2,0) [fill = blue!20, draw = blue, circle] {Zoom red};

  \draw [green!50!black] (4,0) rectangle (6,2);
  \begin{scope}[name = me, view = {(0,0) (2,2) at (4,0) (6,2)}]
    \draw [blue] (5mm,5mm) rectangle (15mm,20mm);
    \node at (5mm,5mm) [circle, fill=blue, text=white] {blue};

    \draw [red] (10mm,10mm) rectangle (15mm,15mm);
    \node at (10mm,10mm) [circle, fill=red, text=white, font=\tiny] {red};
  \end{scope}
}
```

116.6 Commands for Specifying Timing: Beginnings and Endings

Using the `entry` key repeatedly, you specify a timeline: The ⟨*time*⟩ used with the first use of the `entry` key in a timeline is the start time and the ⟨*time*⟩ in the last `entry` key is the stop time. However, this leaves open then question of when the whole timeline is to be started: The moment the document is opened? When the

page is displayed? When the user scrolls to the to-be-animated object? When some other object is clicked? The key `begin`, and also the key `end`, allow you to specify answers to these questions.

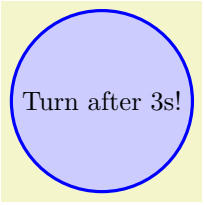
`/pgf/animation/begin`=⟨*time*⟩ (no default)

This key specifies when the "moment `0s`" should be relative to the moment when the current graphic is first displayed. You can use this key multiple times, in this case the timeline is restarted for each of the times specified (if it is already running, it will be reset). If no `begin` key is given at all, the effect is the same as if `begin=0s` had been specified.

It is permissible to set ⟨*time*⟩ to a negative value.

`/pgf/animation/end`=⟨*time*⟩ (no default)

This key will truncate the timeline so that it ends ⟨*time*⟩ after the display of the graphic, provided the timeline begins before the specified end time. For instance, if you specify a timeline starting at 2 s and ending at 5 s and you set `begin` to 1 s and `end` to 4 s, the timeline will run, relative to the moment when the graphic is displayed from 3 s to 4 s.

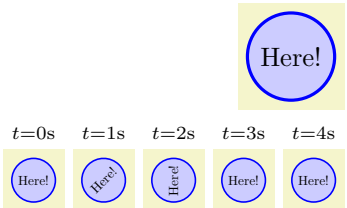



```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin = 2s, end = 4s,
    entry = {1s}{0}, entry = {2s}{90}, entry = {3s}{180}, entry = {4s}{270} }
  \node (node) [fill = blue!20, draw = blue, circle] {Turn after 3s!}; }
```

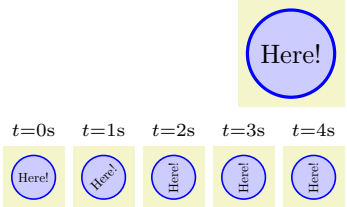
It is not immediately clear what should happen with the attribute of an object when an animation ends: Should it revert to its original value "as if there had never been an animation" or should it "stay at the last value"? The following key governs what should happen:

`/pgf/animation/freeze at end`=⟨*true or false*⟩ (default `true`, initially `false`)

When set to `true`, whenever a timeline ends (either because the last time of timeline has been reached or because an `end` or `end of` key have ended it prematurely), the last value the attribute had because of the animation "stays put". When set to `false`, which is the initial value, once an animation ends, its effect will be removed "as if it never happened".




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, freeze at end = false,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Here!}; }
```




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, freeze at end,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Here!}; }
```

Instead of specifying the beginning of the timeline relative to the moment to to-be-animated graphic is displayed, you can also set the "moment `0s`" to the moment a specific *event* happens using the following key:

`/pgf/animation/begin on`=⟨*options*⟩ (no default)

Has the same effect as `/tikz/animate/option/begin on`, see Section 26.5.2.

When you use `begin on` to start an animation when a certain event is triggered, it is not clear what should happen when the event is triggered *again*. Should this be ignored completely? Should it only be ignored while the animation is running? The following key allows you to specify when should happen:

/pgf/animation/restart=⟨*choice*⟩ (default `true`)

Has the same effect as `/tikz/animate/option/restart`, see Section 26.5.2.

Just like `begin on` specifies when a timeline begins relative to some event, the `end on` allows you to stop is early when some event happens:

/pgf/animation/end on=⟨*options*⟩ (no default)

Works exactly like `begin on`, one possible end of the timeline is specified using the ⟨*options*⟩.

116.7 Commands for Specifying Timing: Repeats

Normally, a timeline is displayed once and then ends. You can, however, request that the timeline should be repeated a certain number of times or indefinitely.

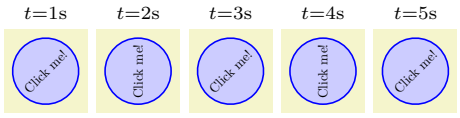
/pgf/animation/repeats=⟨*specification*⟩ (no default)

Use this key to specify that the timeline animation should repeat at the end. The ⟨*specification*⟩ must consist of two parts, each of which may be empty. The first part is one of the following:

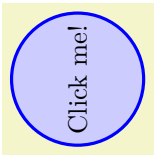
- Empty, in which case the timeline repeats forever.



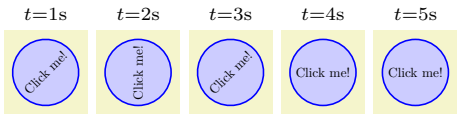

```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, repeats,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```



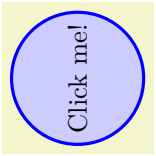
- A ⟨*number*⟩ (like `2` or `3.25`), in which case the timeline repeats ⟨*number*⟩ times.



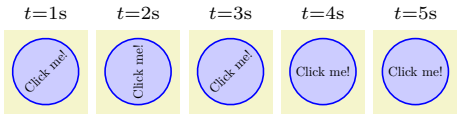

```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, repeats = 1.75,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```



- The text "`for` ⟨*time*⟩" (like `for 2s` or `for 300ms`), in which case the timeline repeats however often necessary so that it stops exactly after ⟨*time*⟩.




```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, repeats = for 3.5s,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```



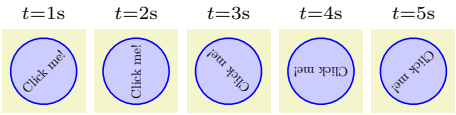
The second part of the specification must be one of the following:

- Empty, in which case each time the timeline is restarted, the attribute's value undergoes the same series of values it did previously.
- The text `accumulating`. This has the effect that each time the timeline is restarted, the attribute values specified by the timeline are *added* to the value from the previous iteration(s). A typical example is an animation that shifts a scope by, say, 1 cm over a time of 1 s. Now, if you repeat this

five times, normally the scope will shift 1 cm for 1 s then “jump back”, shift again, jump back, and so on for five times. In contrast, when the repeats are accumulating, the scope will move by 5 cm over 5 s in total.

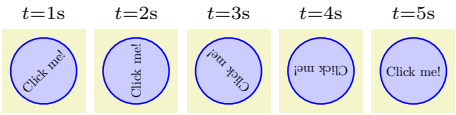



```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, repeats = accumulating,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```






```
\usepgfmodule {animations}
\tikz [very thick] {
  \pgfanimateattribute{rotate}{
    whom = node, begin on = {click}, repeats = 2 accumulating,
    entry = {0s}{0}, entry = {2s}{90} }
  \node (node) [fill = blue!20, draw = blue, circle] {Click me!}; }
```



/pgf/animation/repeat=⟨*specification*⟩ (no default)

This is an alias for `repeats`.

117 Adding libraries to pgf: temporary registers

This section is intended for those who like to write libraries to extend PGF. Of course, this requires a good deal of knowledge about TEX-programming and the structure of the PGF basic layer. Besides, one will encounter the need of temporary variables and, especially, temporary TEX registers. This section describes how to use a set of pre-allocated temporary registers of the basic layer without needing to allocate more of them.

A part of these internals are already mentioned in section 101.7, but the basic layer provides more temporaries than `\pgf@x` and `\pgf@y`.

Internal dimen register `\pgf@x`
Internal dimen register `\pgf@y`

These registers are used to process point coordinates in the basic layer of PGF, see section 101.7. After a `\pgfpoint...` command, they contain the final x and y coordinate, respectively.

The values of `\pgf@x` and `\pgf@y` are set *globally* in contrast to other available PGF registers. You should never assume anything about their value unless the context defines them explicitly.

Please prefer the `\pgf@xa`, `\pgf@xb`, ... registers for temporary dimen registers unless you are writing point coordinate commands.

Internal dimen register `\pgf@xa`
Internal dimen register `\pgf@xb`
Internal dimen register `\pgf@xc`
Internal dimen register `\pgf@ya`
Internal dimen register `\pgf@yb`
Internal dimen register `\pgf@yc`

Temporary registers for TEX dimensions which can be modified freely. Just make sure changes occur only within TEX groups.

Attention: PGF uses these registers to perform path operations. For reasons of efficiency, path commands do not always guard them. As a consequence, the code

```
\pgfpointadd{\pgfpoint{\pgf@xa}{\pgf@ya}}{\pgfpoint{\pgf@xb}{\pgf@yb}}
```

may fail: Inside `\pgfpointadd`, the `\pgf@xa` and friend registers might be modified. In particular, it might happen that `\pgf@xb` is changed before `\pgfpoint{\pgf@xb}{\pgf@yb}` is evaluated. The right thing to do would be to first expand everything using `\edef` and process the values afterwards, resulting in unnecessary expensive operations. Of course, one can avoid this by simply looking into the source code of `\pgfpointadd` to see which registers are used.

Internal dimen register `\pgfutil@tempdima`
Internal dimen register `\pgfutil@tempdimb`

Further multi-purpose temporary dimen registers. For LATEX, these registers are already allocated as `\@tempdima` and `\@tempdimb` and are simply `\let` to the `\pgfutil@...` names.

Internal count register `\c@pgf@counta`
Internal count register `\c@pgf@countb`
Internal count register `\c@pgf@countc`
Internal count register `\c@pgf@countd`

These multiple-purpose count registers are used throughout PGF to perform integer computations. Feel free to use them as well, just make sure changes are scoped by local TEX groups.

Internal openout handle `\w@pgf@writea`

An `\openout` handle which is used to generate complete output files within locally scoped parts of PGF (for example, to interact with `gnuplot`). You should always use `\immediate` in front of output operations involving `\w@pgf@writea` and you should always close the file before returning from your code.

```
\immediate\openout\w@pgf@writea=myfile.dat
\immediate\write\w@pgf@writea{...}%
\immediate\write\w@pgf@writea{...}%
\immediate\closeout\w@pgf@writea%
```

Internal openin handle \r@pgf@reada

An \openin handle which is used to read files within locally scoped parts of PGF, for example to check if a file exists or to read data files. You should always use \immediate in front of output operations involving \r@pgf@writea and you should always close the file before returning from your code.

```
\immediate\openin\r@pgf@reada=myfile.dat
% do something with \macro
\ifeof\r@pgf@reada
    % end of file or it doesn't exist
\else
    % loop or whatever
    \immediate\read\r@pgf@reada to\macro
    ...
\fi
\immediate\closein\r@pgf@reada
```

Internal box \pgfutil@tempboxa

A box for temporary use inside of local TEX scopes. For LATEX, this box is the same as the already pre-allocated \@tempboxa.

118 Quick Commands

This section explains the "quick" commands of PGF. These commands are executed more quickly than the normal commands of PGF, but offer less functionality. You should use these commands only if you either have a very large number of commands that need to be processed or if you expect your commands to be executed very often.

118.1 Quick Coordinate Commands

`\pgfqpoint{`⟨x⟩`}{`⟨y⟩`}`

This command does the same as `\pgfpoint`, but ⟨x⟩ and ⟨y⟩ must be simple dimensions like `1pt` or `1cm`. Things like `2ex` or `2cm+1pt` are not allowed.

`\pgfqpointxy{`⟨s_x⟩`}{`⟨s_y⟩`}`

This command does the same as `\pgfpointxy`, but ⟨s_x⟩ and ⟨s_y⟩ must be simple numbers without unit, like `1.234` or `5.0`. Mathematical expressions or units are not allowed.

`\pgfqpointxyz{`⟨s_x⟩`}{`⟨s_y⟩`}{`⟨s_z⟩`}`

As `\pgfqpointxy`, but for three-dimensional coordinates. Any argument needs to be a number without unit.

`\pgfqpointscale{`⟨*factor*⟩`}{`⟨*coordinate*⟩`}`

As `\pgfpointscale`, but `{`⟨*factor*⟩`}` must be a simple number without unit, as for the other "quick" commands.

118.2 Quick Path Construction Commands

The difference between the quick and the normal path commands is that the quick path commands

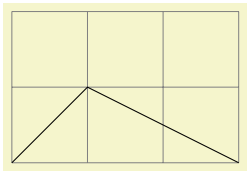
- do not keep track of the bounding boxes,
- do not allow you to arc corners,
- do not apply coordinate transformations.

However, they do use the soft-path subsystem (see Section 121 for details), which allows you to mix quick and normal path commands arbitrarily.

All quick path construction commands start with `\pgfpathq`.

`\pgfpathqmoveto{`⟨*x dimension*⟩`}{`⟨*y dimension*⟩`}`

Either starts a path or starts a new part of a path at the coordinate (⟨*x dimension*⟩, ⟨*y dimension*⟩). The coordinate is *not* transformed by the current coordinate transformation matrix. However, any low-level transformations apply.



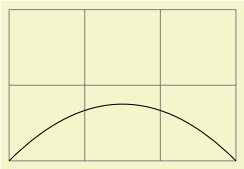
```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgftransformxshift{1cm}
  \pgfpathqmoveto{0pt}{0pt} % not transformed
  \pgfpathqlineto{1cm}{1cm} % not transformed
  \pgfpathlineto{\pgfpoint{2cm}{0cm}}
  \pgfusepath{stroke}
\end{tikzpicture}
```

`\pgfpathqlineto{`⟨*x dimension*⟩`}{`⟨*y dimension*⟩`}`

The quick version of the line-to operation.

`\pgfpathqcurveto{`⟨s_x^1⟩`}{`⟨s_y^1⟩`}{`⟨s_x^2⟩`}{`⟨s_y^2⟩`}{`⟨t_x⟩`}{`⟨t_y⟩`}`

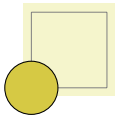
The quick version of the curve-to operation. The first support point is (s_x^1, s_y^1), the second support point is (s_x^2, s_y^2), and the target is (t_x, t_y).



```
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (3,2);
  \pgfpathqmoveto{0pt}{0pt}
  \pgfpathqcurveto{1cm}{1cm}{2cm}{1cm}{3cm}{0cm}
  \pgfusepath{stroke}
\end{tikzpicture}
```

\pgfpathqcircle{⟨*radius*⟩}

Adds a radius around the origin of the given ⟨*radius*⟩. This command is orders of magnitude faster than \pgfpathcircle{\pgfpointorigin}{⟨*radius*⟩}.



```
\colorlet{examplefill}{yellow!80!black}
\begin{tikzpicture}
  \draw[help lines] (0,0) grid (1,1);
  \pgfpathqcircle{10pt}
  \pgfsetfillcolor{examplefill}
  \pgfusepath{stroke,fill}
\end{tikzpicture}
```

118.3 Quick Path Usage Commands

The quick path usage commands perform similar tasks as \pgfusepath, but they

- do not add arrows,
- do not modify the path in any way, in particular,
- ends are not shortened,
- corners are not replaced by arcs.

Note that you *have to* use the quick versions in the code of arrow tip definitions since, inside these definition, you obviously do not want arrows to be drawn.

\pgfusepathqstroke

Strokes the path without further ado. No arrows are drawn, no corners are arced.



```
\begin{pgfpicture}
  \pgfpathqcircle{5pt}
  \pgfusepathqstroke
\end{pgfpicture}
```

\pgfusepathqfill

Fills the path without further ado.

\pgfusepathqfillstroke

Fills and then strokes the path without further ado.

\pgfusepathqclip

Clips all subsequent drawings against the current path. The path is not processed.

118.4 Quick Text Box Commands

\pgfqbox{⟨*box number*⟩}

This command inserts a TEX box into a {pgfpicture} by "escaping" to TEX, inserting the box number ⟨*box number*⟩ at the origin, and then returning to the typesetting the picture.

\pgfqboxsynced{⟨*box number*⟩}

This command works similarly to the \pgfqbox command. However, before inserting the text in ⟨*box number*⟩, the current coordinate transformation matrix is applied to the current canvas transformation matrix (is it "synced" with this matrix, hence the name).

Thus, this command basically has the same effect as if you first called `\pgflowlevelsynccm` followed by `\pgfqbox`. However, this command will use `\hskip` and `\raise` commands for the "translational part" of the coordinate transformation matrix, instead of adding the translational part to the current canvas transformation matrix directly. Both methods have the same effect (box ⟨*box number*⟩ is translated to where it should be), but the method used by `\pgfqboxsynced` ensures that hyperlinks are placed correctly. Note that scaling and rotation will not (cannot, even) apply to hyperlinks.

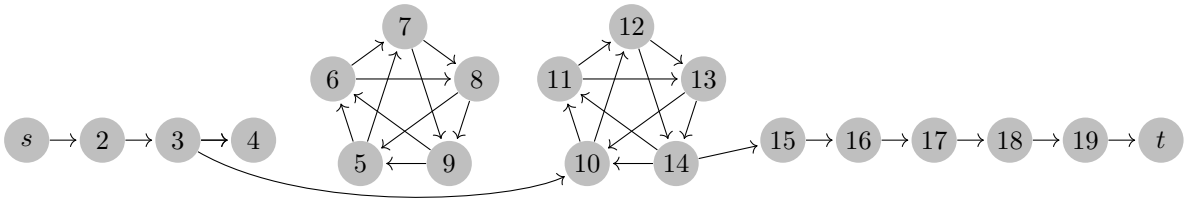
Part X
The System Layer

by Till Tantau

This part describes the low-level interface of PGF, called the *system layer*. This interface provides a complete abstraction of the internals of the underlying drivers.

Unless you intend to port PGF to another driver or unless you intend to write your own optimized frontend, you need not read this part.

In the following it is assumed that you are familiar with the basic workings of the `graphics` package and that you know what TEX-drivers are and how they work.




```
\begin{tikzpicture}
  [shorten >=1pt,->,
   vertex/.style={circle,fill=black!25,minimum size=17pt,inner sep=0pt}]

  \foreach \name/\x in {s/1, 2/2, 3/3, 4/4, 15/11, 16/12, 17/13, 18/14, 19/15, t/16}
    \node[vertex] (G-\name) at (\x,0) {$\name$};

  \foreach \name/\angle/\text in {P-1/234/5, P-2/162/6, P-3/90/7, P-4/18/8, P-5/-54/9}
    \node[vertex,xshift=6cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \name/\angle/\text in {Q-1/234/10, Q-2/162/11, Q-3/90/12, Q-4/18/13, Q-5/-54/14}
    \node[vertex,xshift=9cm,yshift=.5cm] (\name) at (\angle:1cm) {$\text$};

  \foreach \from/\to in {s/2,2/3,3/4,3/4,15/16,16/17,17/18,18/19,19/t}
    \draw (G-\from) -- (G-\to);

  \foreach \from/\to in {1/2,2/3,3/4,4/5,5/1,1/3,2/4,3/5,4/1,5/2}
    { \draw (P-\from) -- (P-\to); \draw (Q-\from) -- (Q-\to); }

  \draw (G-3) .. controls +(-30:2cm) and +(-150:1cm) .. (Q-1);
  \draw (Q-5) -- (G-15);
\end{tikzpicture}
```

119 Design of the System Layer

119.1 Driver Files

The PGF system layer mainly consists of a large number of commands starting with `\pgfsys@`. These commands will be called *system commands* in the following. The higher layers "interface" with the system layer by calling these commands. The higher layers should never use `\special` commands directly or even check whether `\pdfoutput` is defined. Instead, all drawing requests should be "channeled" through the system commands.

The system layer is loaded and set up by the following package:

```
\usepackage{pgfsys} % LaTeX
\input{pgfsys.tex} % plain TeX
\usemodule[pgfsys]  % ConTeXt
```

This file provides "default implementations" of all system commands, but most simply produce a warning that they are not implemented. The actual implementations of the system commands for a particular driver like, say, `pdftex` reside in files called `pgfsys-xxxx.sty`, where `xxxx` is the driver name. These will be called *driver files* in the following.

When `pgfsys.sty` is loaded, it will try to determine which driver is used by loading `pgf.cfg`. This file should set up the macro `\pgfsysdriver` appropriately. The `pgfsys.sty` will input the appropriate `pgfsys-`⟨*drivername*⟩`.sty`.

`\pgfsysdriver`

This macro should expand to the name of the driver to be used by `pgfsys`. The default from `pgf.cfg` is `pgfsys-\Gin@driver`. This is very likely to be correct if you are using LaTeX. For plain TeX, the macro will be set to `pgfsys-pdftex.def` if `pdftex` is used and to `pgfsys-dvips.def` otherwise.

File `pgf.cfg`

This file should set up the command `\pgfsysdriver` correctly. If `\pgfsysdriver` is already set to some value, the driver normally should not change it. Otherwise, it should make a "good guess" at which driver will be appropriate.

The currently supported backend drivers are discussed in Section 10.2.

119.2 Common Definition Files

Some drivers share many `\pgfsys@` commands. For the reason, files defining these "common" commands are available. These files are *not* usable alone.

File `pgfsys-common-postscript`

This file defines some `\pgfsys@` commands so that they produce appropriate PostScript code.

File `pgfsys-common-pdf`

This file defines some `\pgfsys@` commands so that they produce appropriate PDF code.

120 Commands of the System Layer

120.1 Beginning and Ending a Stream of System Commands

A "user" of the PGF system layer (like the basic layer or a frontend) will interface with the system layer by calling a stream of commands starting with `\pgfsys@`. From the system layer's point of view, these commands form a long stream. Between calls to the system layer, control goes back to the user.

The driver files implement system layer commands by inserting `\special` commands that implement the desired operation. For example, `\pgfsys@stroke` will be mapped to `\special{pdf: S}` by the driver file for `pdftex`.

For many drivers, when such a stream of specials starts, it is necessary to install an appropriate transformation and perhaps perform some more bureaucratic tasks. For this reason, every stream will start with a `\pgfsys@beginpicture` and will end with a corresponding ending command.

`\pgfsys@beginpicture`

Called at the beginning of a `{pgfpicture}`. This command should "set up things".

Most drivers will need to implement this command.

`\pgfsys@endpicture`

Called at the end of a `{pgfpicture}`.

Most drivers will need to implement this command.

`\pgfsys@typesetpicturebox{`⟨*box*⟩`}`

Called *after* a `{pgfpicture}` has been typeset. The picture will have been put in box ⟨*box*⟩. This command should insert the box into the normal text. The box ⟨*box*⟩ will still be a "raw" box that contains only the `\special`'s that make up the description of the picture. The job of this command is to resize and shift ⟨*box*⟩ according to the baseline shift and the size of the box.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@beginpurepicture`

This version of the `\pgfsys@beginpicture` picture command can be used for pictures that are guaranteed not to contain any escaped boxes (see below). In this case, a driver might provide a more compact version of the command.

This command has a default implementation and need not be implemented by a driver file.

`\pgfsys@endpurepicture`

Called at the end of a "pure" `{pgfpicture}`.

This command has a default implementation and need not be implemented by a driver file.

Inside a stream it is sometimes necessary to "escape" back into normal typesetting mode; for example to insert some normal text, but with all of the current transformations and clippings being in force. For this escaping, the following command is used:

`\pgfsys@hbox{`⟨*box number*⟩`}`

Called to insert a (horizontal) TEX box inside a `{pgfpicture}`.

Most drivers will need to (re-)implement this command.

`\pgfsys@hboxsynced{`⟨*box number*⟩`}`

Called to insert a (horizontal) TEX box inside a `{pgfpicture}`, but with the current coordinate transformation matrix synced with the canvas transformation matrix.

This command should do the same as if you used `\pgflowlevelsynccm` followed by `\pgfsys@hbox`. However, the default implementation of this command will use a "TEX-translation" for the translation part of the transformation matrix. This will ensure that hyperlinks "survive" at least translations. On the other hand, a driver may choose to revert to a simpler implementation. This is done, for example, for the SVG implementation, where a TEX-translation makes no sense.

\pgfsys@pictureboxsynced{⟨*box number*⟩}

Basically, this should do the same as doing a (scoped) low level sync followed by inserting the box ⟨*box number*⟩ directly into the output stream. However, the default implementation uses `\pgfsys@hboxsynced` in conjunction with `\pgfsys@beginpicture` to ensure that, if possible, hyperlinks survive in PDFs. Drivers that are sensitive to picture-in-picture scopes should replace this implementation by

```
\pgfsys@beginscope\pgflowlevelsynccm\box#1\pgfsys@endscope
```

120.2 Scoping System Commands

The scoping commands are used to keep changes of the graphics state local.

\pgfsys@beginscope

Saves the current graphic state on a graphic state stack. All changes to the graphic state parameters mentioned for `\pgfsys@stroke` and `\pgfsys@fill` will be local to the current graphic state and the old values will be restored after `\pgfsys@endscope` is used.

Warning: PDF and PostScript differ with respect to the question of whether the current path is part of the graphic state or not. For this reason, you should never use this command unless the path is currently empty. For example, it might be a good idea to use `\pgfsys@discardpath` prior to calling this command.

This command is protocolled, see Section 122.

\pgfsys@endscope

Restores the last saved graphic state.

This command is protocolled, see Section 122.

120.3 Path Construction System Commands

\pgfsys@moveto{⟨x⟩}{⟨y⟩}

This command is used to start a path at a specific point (x, y) or to move the current point of the current path to (x, y) without drawing anything upon stroking (the current path is "interrupted").

Both ⟨x⟩ and ⟨y⟩ are given as TEX dimensions. It is the driver's job to transform these to the coordinate system of the backend. Typically, this means converting the TEX dimension into a dimensionless multiple of $\frac{1}{72}$in. The function `\pgf@sys@bp` helps with this conversion.

Example: Draw a line from (10pt, 10pt) to the origin of the picture.

```
\pgfsys@moveto{10pt}{10pt}
\pgfsys@lineto{0pt}{0pt}
\pgfsys@stroke
```

This command is protocolled, see Section 122.

\pgfsys@lineto{⟨x⟩}{⟨y⟩}

Continue the current path to (x, y) with a straight line.

This command is protocolled, see Section 122.

\pgfsys@curveto{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x_3⟩}{⟨y_3⟩}

Continue the current path to (x_3, y_3) with a Bézier curve that has the two control points (x_1, y_1) and (x_2, y_2).

Example: Draw a good approximation of a quarter circle:

```
\pgfsys@moveto{10pt}{0pt}
\pgfsys@curveto{10pt}{5.55pt}{5.55pt}{10pt}{0pt}{10pt}
\pgfsys@stroke
```

This command is protocolled, see Section 122.

\pgfsys@rect{⟨x⟩}{⟨y⟩}{⟨width⟩}{⟨height⟩}

Append a rectangle to the current path whose lower left corner is at (x, y) and whose width and height in big points are given by ⟨*width*⟩ and ⟨*height*⟩.

This command can be "mapped back" to \pgfsys@moveto and \pgfsys@lineto commands, but it is included since PDF has a special, quick version of this command.

This command is protocolled, see Section 122.

\pgfsys@closepath

Close the current path. This results in joining the current point of the path with the point specified by the last \pgfsys@moveto operation. Typically, this is preferable over using \pgfsys@lineto to the last point specified by a \pgfsys@moveto, since the line starting at this point and the line ending at this point will be smoothly joined by \pgfsys@closepath.

Example: Consider

```
\pgfsys@moveto{0pt}{0pt}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@closepath
\pgfsys@stroke
```

and

```
\pgfsys@moveto{0bp}{0bp}
\pgfsys@lineto{10bp}{10bp}
\pgfsys@lineto{0bp}{10bp}
\pgfsys@lineto{0bp}{0bp}
\pgfsys@stroke
```

The difference between the above will be that in the second triangle the corner at the origin will be wrong; it will just be the overlay of two lines going in different directions, not a sharp pointed corner.

This command is protocolled, see Section 122.

120.4 Canvas Transformation System Commands

\pgfsys@transformcm{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨e⟩}{⟨f⟩}

Perform a concatenation of the canvas transformation matrix with the matrix given by the values ⟨*a*⟩ to ⟨*f*⟩, see the PDF or PostScript manual for details. The values ⟨*a*⟩ to ⟨*d*⟩ are dimensionless factors, ⟨*e*⟩ and ⟨*f*⟩ are TEX dimensions

Example: \pgfsys@transformcm{1}{0}{0}{1}{1cm}{1cm}.

This command is protocolled, see Section 122.

\pgfsys@transformshift{⟨*x displacement*⟩}{⟨*y displacement*⟩}

This command will change the origin of the canvas to (x, y).

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 122.

\pgfsys@transformxyscale{⟨*x scale*⟩}{⟨*y scale*⟩}

This command will scale the canvas (and everything that is drawn) by a factor of ⟨*x scale*⟩ in the x-direction and ⟨*y scale*⟩ in the y-direction. Note that this applies to everything, including lines. So a scaled line will have a different width and may even have a different width when going along the x-axis and when going along the y-axis, if the scaling is different in these directions. Usually, you do not want this.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 122.

\pgfsys@viewboxmeet{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨x'_1⟩}{⟨y'_1⟩}{⟨x'_2⟩}{⟨y'_2⟩}

Starts a "view box" scope, which must be ended using \pgfsys@endviewbox later on (with matching scopes).

The effect of this command is as follows: Consider the rectangles R with lower left corner (x_1, y_1) and upper right corner (x_2, y_2) and R' with corners (x'_1, y'_1) and (x'_2, y'_2). The command will install a canvas translation and uniform scaling such that R' then has the same center as R and additionally, has maximum size such that it still fits inside R. (Think of this as "viewing" R' through R such that the aspect ratio is kept.)

This command has a default implementation. Its main purpose is to allow animations of the view box; for static drawings it is better to compute the necessary transformations directly.

`\pgfsys@viewboxslice{`$\langle x_1 \rangle$`}{`$\langle y_1 \rangle$`}{`$\langle x_2 \rangle$`}{`$\langle y_2 \rangle$`}{`$\langle x'_1 \rangle$`}{`$\langle y'_1 \rangle$`}{`$\langle x'_2 \rangle$`}{`$\langle y'_2 \rangle$`}`

Works like the previous command, but now R' has minimal size such that it encompasses all of R.

`\pgfsys@endviewbox`

Ends a viewbox previously started using `\pgfsys@viewboxmeet` or the `...slice` variant.

120.5 Stroking, Filling, and Clipping System Commands

`\pgfsys@stroke`

Stroke the current path (as if it were drawn with a pen). A number of graphic state parameters influence this, which can be set using appropriate system commands described later.

Line width The "thickness" of the line. A width of 0 is the thinnest width renderable on the device. On a high-resolution printer this may become invisible and should be avoided. A good choice is 0.4pt, which is the default.

Stroke color This special color is used for stroking. If it is not set, the current color is used.

Cap The cap describes how the endings of lines are drawn. A round cap adds a little half circle to these endings. A butt cap ends the lines exactly at the end (or start) point without anything added. A rectangular cap ends the lines like the butt cap, but the lines protrude over the endpoint by the line thickness. (See also the PDF manual.) If the path has been closed, no cap is drawn.

Join This describes how a bend (a join) in a path is rendered. A round join draws bends using small arcs. A bevel join just draws the two lines and then fills the join minimally so that it becomes convex. A miter join extends the lines so that they form a single sharp corner, but only up to a certain miter limit. (See the PDF manual once more.)

Dash The line may be dashed according to a dashing pattern.

Clipping area If a clipping area is established, only those parts of the path that are inside the clipping area will be drawn.

In addition to stroking a path, the path may also be used for clipping after it has been stroked. This will happen if the `\pgfsys@clipnext` is used prior to this command, see there for details.

This command is protocolled, see Section 122.

`\pgfsys@closestroke`

This command should have the same effect as first closing the path and then stroking it.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 122.

`\pgfsys@fill`

This command fills the area surrounded by the current path. If the path has not yet been closed, it is closed prior to filling. The path itself is not stroked. For self-intersecting paths or paths consisting of multiple parts, the nonzero winding number rule is used to determine whether a point is inside or outside the path, except if `\ifpgfsys@eorule` holds – in which case the even-odd rule should be used. (See the PDF or PostScript manual for details.)

The following graphic state parameters influence the filling:

Interior rule If `\ifpgfsys@eorule` is set, the even-odd rule is used, otherwise the non-zero winding number rule.

Fill color If the fill color is not especially set, the current color is used.

Clipping area If a clipping area is established, only those parts of the filling area that are inside the clipping area will be drawn.

In addition to filling the path, the path will also be used for clipping if `\pgfsys@clipnext` is used prior to this command.

This command is protocolled, see Section 122.

`\pgfsys@fillstroke`

First, the path is filled, then the path is stroked. If the fill and stroke colors are the same (or if they are not specified and the current color is used), this yields almost the same as a `\pgfsys@fill`. However, due to the line thickness of the stroked path, the fill-stroked area will be slightly larger.

In addition to stroking and filling the path, the path will also be used for clipping if `\pgfsys@clipnext` is used prior to this command.

This command is protocolled, see Section 122.

`\pgfsys@discardpath`

Normally, this command should "throw away" the current path. However, after `\pgfsys@clipnext` has been called, the current path should subsequently be used for clipping. See `\pgfsys@clipnext` for details.

This command is protocolled, see Section 122.

`\pgfsys@clipnext`

This command should be issued after a path has been constructed, but before it has been stroked and/or filled or discarded. When the command is used, the next stroking/filling/discarding command will first be executed normally. Then, afterwards, the just-used path will be used for subsequent clipping. If there has already been a clipping region, this region is intersected with the new clipping path (the clipping cannot get bigger). The nonzero winding number rule is used to determine whether a point is inside or outside the clipping area or the even-odd rule, depending on whether `\ifpgfsys@eorule` holds.

120.6 Graphic State Option System Commands

`\pgfsys@setlinewidth{`⟨*width*⟩`}`

Sets the width of lines, when stroked, to ⟨*width*⟩, which must be a TEX dimension.

This command is protocolled, see Section 122.

`\pgfsys@buttcap`

Sets the cap to a butt cap. See `\pgfsys@stroke`.

This command is protocolled, see Section 122.

`\pgfsys@roundcap`

Sets the cap to a round cap. See `\pgfsys@stroke`.

This command is protocolled, see Section 122.

`\pgfsys@rectcap`

Sets the cap to a rectangular cap. See `\pgfsys@stroke`.

This command is protocolled, see Section 122.

`\pgfsys@miterjoin`

Sets the join to a miter join. See `\pgfsys@stroke`.

This command is protocolled, see Section 122.

`\pgfsys@setmiterlimit{`⟨*factor*⟩`}`

Sets the miter limit of lines to ⟨*factor*⟩. See the PDF or PostScript for details on what the miter limit is.

This command is protocolled, see Section 122.

`\pgfsys@roundjoin`

Sets the join to a round join. See `\pgfsys@stroke`.

This command is protocolled, see Section 122.

\pgfsys@beveljoin

Sets the join to a bevel join. See \pgfsys@stroke.

This command is protocolled, see Section 122.

\pgfsys@setdash{⟨*pattern*⟩}{⟨*phase*⟩}

Sets the dashing pattern. ⟨*pattern*⟩ should be a list of TEX dimensions separated by commas. ⟨*phase*⟩ should be a single dimension.

Example: \pgfsys@setdash{3pt,3pt}{0pt}

The list of values in ⟨*pattern*⟩ is used to determine the lengths of the "on" and "off" phases of the dashing. For example, if ⟨*pattern*⟩ is 3bp,4bp, then the dashing pattern is "3bp on followed by 4bp off, followed by 3bp on, followed by 4bp off, and so on". A pattern of .5pt,4pt,3pt,1.5pt means ".5pt on, 4pt off, 3pt on, 1.5pt off, .5pt on, …" If the number of entries is odd, the last one is used twice, so 3pt means "3pt on, 3pt off, 3pt on, 3pt off, …" An empty list means "always on".

The second argument determines the "phase" of the pattern. For example, for a pattern of 3bp,4bp and a phase of 1bp, the pattern would start: "2bp on, 4bp off, 3bp on, 4bp off, 3bp on, 4bp off, …"

This command is protocolled, see Section 122.

\ifpgfsys@eorule

Determines whether the even odd rule is used for filling and clipping or not.

120.7 Color System Commands

The PGF system layer provides a number of system commands for setting colors. These command coexist with commands from the color and xcolor package, which perform similar functions. However, the color package does not support having two different colors for stroking and filling, which is a useful feature that is supported by PGF. For this reason, the PGF system layer offers commands for setting these colors separately. Also, plain TEX profits from the fact that PGF can set colors.

For PDF, implementing these color commands is easy since PDF supports different stroking and filling colors directly. For PostScript, a more complicated approach is needed in which the colors need to be stored in special PostScript variables that are set whenever a stroking or a filling operation is done.

\pgfsys@color@rgb{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for stroking and filling operations to the given red/green/blue tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@rgb@stroke{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for stroking operations to the given red/green/blue tuple (numbers between 0 and 1).

Example: Make stroked text dark red: \pgfsys@color@rgb@stroke{0.5}{0}{0}

The special stroking color is only used if the stroking color has been set since the last \color or \pgfsys@color@... command. Thus, each \color command will reset both the stroking and filling colors by calling \pgfsys@color@reset.

This command is protocolled, see Section 122.

\pgfsys@color@rgb@fill{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the color used for filling operations to the given red/green/blue tuple (numbers between 0 and 1). This color may be different from the stroking color.

This command is protocolled, see Section 122.

\pgfsys@color@cmyk{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for stroking and filling operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@cmyk@stroke{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for stroking operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@cmyk@fill{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Sets the color used for filling operations to the given cmyk tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@cmy{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for stroking and filling operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@cmy@stroke{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for stroking operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@cmy@fill{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Sets the color used for filling operations to the given cmy tuple (numbers between 0 and 1).

This command is protocolled, see Section 122.

\pgfsys@color@gray{⟨*black*⟩}

Sets the color used for stroking and filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 122.

\pgfsys@color@gray@stroke{⟨*black*⟩}

Sets the color used for stroking operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 122.

\pgfsys@color@gray@fill{⟨*black*⟩}

Sets the color used for filling operations to the given black value, where 0 means black and 1 means white.

This command is protocolled, see Section 122.

\pgfsys@color@reset

This command will be called when the \color command is used. It should purge any internal settings of stroking and filling color. After this call, till the next use of a command like \pgfsys@color@rgb@fill, the current color installed by the \color command should be used.

If the TEX-if \pgfsys@color@reset@inorder is set to true, this command may "assume" that any call to a color command that sets the fill or stroke color came "before" the call to this command and may try to optimize the output accordingly.

An example of an incorrect "out of order" call would be using \pgfsys@color@reset at the beginning of a box that is constructed using \setbox. Then, when the box is constructed, no special fill or stroke color might be in force. However, when the box is later on inserted at some point, a special fill color might already have been set. In this case, this command is not guaranteed to reset the color correctly.

\pgfsys@color@reset@inordertrue

Sets the optimized "in order" version of the color resetting. This is the default.

\pgfsys@color@reset@inorderfalse

Switches off the optimized color resetting.

\pgfsys@color@unstacked{⟨*LaTeX color*⟩}

This slightly obscure command causes the color stack to be tricked. When called, this command should set the current color to ⟨*LaTeX color*⟩ without causing any change in the color stack.

Example: \pgfsys@color@unstacked{red}

120.8 Pattern System Commands

\pgfsys@declarepattern{⟨*name*⟩}{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}{⟨*x step*⟩}{⟨*y step*⟩}{⟨*a*⟩}{⟨*b*⟩}{⟨*c*⟩}{⟨*d*⟩}{⟨*e*⟩}{⟨*f*⟩}{⟨*code*⟩}{⟨*flag*⟩}

This command declares a new colored or uncolored pattern, depending on whether ⟨*flag*⟩ is `0`, which means uncolored, or `1`, which means colored. Uncolored patterns have no inherent color, the color is provided when they are set. Colored patterns have an inherent color.

The ⟨*name*⟩ is a name for later use when the pattern is to be shown. The pairs (x_1, y_1) and (x_2, y_2) must describe a bounding box of the pattern ⟨*code*⟩.

The tiling step of the pattern is given by ⟨*x step*⟩ and ⟨*y step*⟩.

The parameters ⟨*a*⟩ to ⟨*f*⟩ are entries of the transformation matrix that is applied to the pattern, see `\pgfsys@patternmatrix` for more information.

Example:

```
\pgfsys@declarepattern
    {hori}{-.5pt}{0pt}{.5pt}{3pt}{3pt}{3pt}%
    {1.0}{0.0}{0.0}{1.0}{0.0pt}{0.0pt}%
    {\pgfsys@moveto{0pt}{0pt}\pgfsys@lineto{0pt}{3pt}\pgfsys@stroke}
    {0}
```

\pgfsys@patternmatrix

For convenience PGF defines the transformation matrix that is applied to all patterns defined with `\pgfdeclarepatternformonly` and `\pgfdeclarepatterninherentlycolored` in a macro. This can be used as an extension point for ad-hoc transformation of existing patterns. The default definition is the identity matrix:

```
\def\pgfsys@patternmatrix{{1.0}{0.0}{0.0}{1.0}{0.0pt}{0.0pt}}
```

The entries of the enclosed array `{`⟨*a*⟩`}{`⟨*b*⟩`}{`⟨*c*⟩`}{`⟨*d*⟩`}{`⟨*e*⟩`}{`⟨*f*⟩`}` are entries in the transformation matrix, identified as in the following transformation prescription:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Carrying out the matrix multiplication results in the following system of equations

$$\begin{aligned} x' &= ax + cy + e, \\ y' &= bx + dy + f. \end{aligned}$$

Evidently, the parameters `{`⟨*a*⟩`}` to `{`⟨*d*⟩`}` have to be dimensionless because they are scaling factors, but the parameters `{`⟨*e*⟩`}` and `{`⟨*f*⟩`}` are offsets, therefore they have to carry a unit.

\pgfsys@setpatternuncolored{⟨*name*⟩}{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Sets the fill color to the pattern named ⟨*name*⟩. This pattern must previously have been declared with ⟨*flag*⟩ set to `0`. The color of the pattern is given in the parameters ⟨*red*⟩, ⟨*green*⟩, and ⟨*blue*⟩ in the usual way.

The fill color "pattern" will persist till the next color command that modifies the fill color.

\pgfsys@setpatterncolored{⟨*name*⟩}

Sets the fill color to the pattern named ⟨*name*⟩. This pattern must have been declared with the `1` flag.

120.9 Image System Commands

The system layer provides some commands for image inclusion.

\pgfsys@imagesuffixlist

This macro should expand to a list of suffixes, separated by ':', that will be tried when searching for an image.

Example: `\def\pgfsys@imagesuffixlist{eps:epsi:ps}`

\pgfsys@defineimage

Called, when an image should be defined.

This command does not take any parameters. Instead, certain macros will be preinstalled with appropriate values when this command is invoked. These are:

- \pgf@filename File name of the image to be defined.
- \pgf@imagewidth Will be set to the desired (scaled) width of the image.
- \pgf@imageheight Will be set to the desired (scaled) height of the image.

  If this macro and also the height macro are empty, the image should have its "natural" size.

  If only one of them is specified, the undefined value the image is scaled so that the aspect ratio is kept.

  If both are set, the image is scaled in both directions independently, possibly changing the aspect ratio.

The following macros presumable mostly make sense for drivers that can handle PDF:

- \pgf@imagepage The desired page number to be extracted from a multi-page "image".
- \pgf@imagemask If set, it will be set to `/SMask x 0 R` where `x` is the PDF object number of a soft mask to be applied to the image.
- \pgf@imageinterpolate If set, it will be set to `/Interpolate true` or `/Interpolate false`, indicating whether the image should be interpolated in PDF.

The command should now set up the macro \pgf@image such that calling this macro will result in typesetting the image. Thus, \pgf@image is the "return value" of the command.

This command has a default implementation and need not be implemented by a driver file.

120.10 Shading System Commands

\pgfsys@horishading{⟨*name*⟩}{⟨*height*⟩}{⟨*specification*⟩}

Declares a horizontal shading for later use. The effect of this command should be the definition of a macro called \@pgfshading⟨*name*⟩! (or \csname @pdfshading⟨*name*⟩!\endcsname, to be precise). When invoked, this new macro should insert a shading at the current position.

⟨*name*⟩ is the name of the shading, which is also used in the output macro name. ⟨*height*⟩ is the height of the shading and must be given as a TEX dimension like `2cm` or `10pt`. ⟨*specification*⟩ is a shading color specification as specified in Section 114. The shading specification implicitly fixes the width of the shading.

When \@pgfshading⟨*name*⟩! is invoked, it should insert a box of height ⟨*height*⟩ and the width implicit in the shading declaration.

\pgfsys@vertshading{⟨*name*⟩}{⟨*width*⟩}{⟨*specification*⟩}

Like the horizontal version, only for vertical shadings. This time, the height of the shading is implicit in ⟨*specification*⟩ and the width is given as ⟨*width*⟩.

\pgfsys@radialshading{⟨*name*⟩}{⟨*starting point*⟩}{⟨*specification*⟩}

Declares a radial shading. Like the previous macros, this command should set up the macro \@pgfshading⟨*name*⟩!, which upon invocation should insert a radial shading whose size is implicit in ⟨*specification*⟩.

The parameter ⟨*starting point*⟩ is a PGF point specifying the inner starting point of the shading.

\pgfsys@functionalshading{⟨*name*⟩}{⟨*lower left corner*⟩}⟨*upper right corner*⟩{⟨*type 4 function*⟩}

Declares a shading using a PostScript-like function that provides a color for each point. Like the previous macros, this command should set up the macro \@pgfshading⟨*name*⟩! so that it will produce a box containing the desired shading.

Parameter ⟨*name*⟩ is the name of the shading. Parameter ⟨*type 4 function*⟩ is a Postscript-like function (type 4 function of the PDF specification) as described in Section 3.9.4 of the PDF specification version 1.7. Parameters ⟨*lower left corner*⟩ and ⟨*upper right corner*⟩ are PGF points that specifies the lower left and upper right corners of the shading, respectively.

When ⟨*type 4 function*⟩ is evaluated, the coordinate of the current point will be on the (virtual) PostScript stack in bp units. After the function has been evaluated, the stack should consist of three numbers (not integers! – the Apple PDF renderer is broken in this regard, so add cvrs at the end if needed) that represent the red, green, and blue components of the color.

A buggy function will result is *totally unpredictable chaos* during rendering.

120.11 Transparency System Commands

`\pgfsys@opacity{`⟨*value*⟩`}`

Sets the opacity of all operations, treating stroking and filling as a transparency group. Some drivers support this operations, others do not and set the fill and stroke individually. This difference can only be seen when a path is stroked and filled at the same time: When the drawing and fill opacities are set individually, the effect of filling and drawing a path at the same time is the same as first filling the path and then drawing it. On the other, if the opacity is set using this command, the effect should rather be that same as first filling and then drawing the path without any opacity in an off-screen area and then copying the result to the target area with a homogeneous opacity of ⟨*value*⟩.

Since PDF does not support this form of opacity, this command is only present on the system layer and not supported in the basic layer.

`\pgfsys@stroke@opacity{`⟨*value*⟩`}`

Sets the opacity of stroking operations.

`\pgfsys@fill@opacity{`⟨*value*⟩`}`

Sets the opacity of filling operations.

`\pgfsys@blend@mode{`⟨*value*⟩`}`

Sets the blend mode, see Section 7.2.4 of the PDF Specification, Version 1.7.

`\pgfsys@transparencygroupfrombox{`⟨*box*⟩`}`

This takes a TEX box and converts it into a transparency group. This means that any transparency settings apply to the box as a whole. For instance, if a box contains two overlapping black circles and you draw the box and, thus, the two circles normally with 50% transparency, then the overlap will be darker than the rest. By comparison, if the circles are part of a transparency group, the overlap will get the same color as the rest.

A transparency group can be *isolated* and/or a *knockout* group (see Sections 7.3.4 and 7.3.5 of the PDF Specification Version 1.7). Which of these is the case is dictated by the current settings of the following two ifs, which must be set before the above command is called:

`\ifpgfsys@transparency@group@isolated`

Determines whether a transparency group should be isolated.

`\ifpgfsys@transparency@group@knockout`

Determines whether a transparency group is a knockout group or not.

`\pgfsys@fadingfrombox{`⟨*name*⟩`}{`⟨*box*⟩`}`

Declares the fading ⟨*name*⟩. The ⟨*box*⟩ is a TEX-box. Its content's luminosity determines the opacity of the resulting fading. This means that the lighter a pixel inside the box, the more opaque the fading will be at this position.

`\pgfsys@usefading`⟨*name*⟩`{`⟨*a*⟩`}{`⟨*b*⟩`}{`⟨*c*⟩`}{`⟨*d*⟩`}{`⟨*e*⟩`}{`⟨*f*⟩`}`

Installs a previously declared fading ⟨*name*⟩ in the current graphics state. Afterwards, all drawings will be masked by the fading. The fading should be centered on the origin and have its original size, except that the parameters ⟨*a*⟩ to ⟨*f*⟩ specify a transformation matrix that should be applied additionally to the fading before it is installed. The transformation should not apply to the following graphics, however.

\pgfsys@clipfading

This command has a default implementation and need not be implemented by driver files other than `pgfsys-dvips.def`. The macro is called in

`\pgfsetfadingforcurrentpath` and `\pgfsetfadingforcurrentpathstroked`

of the basic layer, where it invokes the current path for clipping the shading just before installing it as an opacity mask for fading. The default implementation is actually a non-operation, but with `dvips` it is used to clip the fading as described.

\pgfsys@definemask

This command declares a fading (known as a soft mask in this context) based on an image and for usage with images. It works similar to `\pgfsys@defineimage`: Certain macros are set when the command is called. The result should be to set the macro `\pgf@mask` to a pdf object count that can subsequently be used as a transparency mask. The following macros will be set when this command is invoked:

- \pgf@filename File name of the mask to be defined.
- \pgf@maskmatte The so-called matte of the mask (see the PDF documentation for details). The matte is a color specification consisting of 1, 3 or 4 numbers between 0 and 1. The number of numbers depends on the number of color channels in the image (not in the mask!). It will be assumed that the image has been preblended with this color.

120.12 Animation Commands

The animation system layer command (`\pgfsys@anim...`) are described in a separate section, Section 123.

120.13 Object Identification System Commands

The system layer provides commands for adding identification labels (ids) to different objects in a graphic. These can be used for hyperlinking, which is needed for instance in conjunction with animations.

The following "objects" can get an id assigned to them:

1. Graphic scopes (namely when `\pgfsys@begin@idscope` is called),
2. view boxes (namely when `\pgfsys@viewboxmeet` or `\pgfsys@viewboxslice` are called),
3. paths (namely when `\pgfsys@fill`, `\pgfsys@stroke`, and so on are called),
4. text boxes (namely when `\pgfsys@hbox` or `\pgfsys@hboxsynced` is called), and
5. animations (namely when `\pgfsys@animate` is called).

Creating and using ids is a two-step process. First, you create the id using `\pgfsys@new@id`, which stores a fresh id in a macro. You can now pass this id around and clone it. Then, at some point, you wish one of the above objects to actually get this id. For this, you use `\pgfsys@use@id` just *before* the object since this command always influences the *next* object.

The basic id management gets more powerful when you use *id types*. The idea is as follows: In reality, the objects from above do not get assigned only an id, but rather a combination of an id and a type – and you can set the type independently of the id. This is used, for instance, to allow easy access to the different parts of a node in animations: Each node has a single id, but consists of several graphic objects (normally, at least a background path and a text). Each of these uses the same underlying id of the node, but the path has the type `path` (actually `background.path`) while the text has the type `text`. The advantage is that for each node only one id must be stored instead of a great number of the many different possible parts of a node.

\pgfsys@new@id{⟨*macro*⟩}

Creates a new id for later use and stores it in ⟨*macro*⟩. It is an internal text created by the driver and may not be changed or modified.

\pgfsys@use@id{⟨*id*⟩}

"Uses" an id previously created using \pgfsys@new@id. This causes the *next* graphic object to get the ⟨*id*⟩ (not the current one). Once used, the id-type-pair becomes *invalid* and will not be attached to any other graphics objects. It is, however, not an error to try this. If ⟨*id*⟩ is empty, no id-type-pair is attached to the next object.

\pgfsys@use@type{⟨*type*⟩}

Changes the type used with the next graphic object. As mentioned earlier, the id assigned to the next object is actually a pair consisting of the currently used id and the currently used type.

\pgfsys@append@type{⟨*text*⟩}

Appends the ⟨*text*⟩ to the current type.

\pgfsys@push@type

Pushes the current type on a global "stack of types" without opening a TEX scope. The is useful when you temporarily wish to change the type (for instance, by appending something to it), but you cannot create a new scope.

\pgfsys@pop@type

Restores the most recently pushed type.

\pgfsys@begin@idscope

Starts a (graphics) scope whose sole purpose is to assign it an id-type-pair so that it can be referenced later. Note that this command does not always produce a graphics scope: If no id is currently in use or if the id-type-pair has already been used, a graphic scope may or may not be created as defined by the driver (but always a TEX scope). This allows drivers to minimize the number of graphic scopes created.

When an id scope is created, any code that has been "attached" to it using \pgfsys@attach@to@id gets executed, see that command.

Note that \pgfsys@beginscope does not use the current id-type-pair. You need to call this command to attach an id to a group.

\pgfsys@end@idscope

Ends the graphics id scope started by \pgfsys@end@idscope. It must nest correctly with other graphic scopes and TEX scopes.

\pgfsys@attach@to@id{⟨*id*⟩}{⟨*type*⟩}{⟨*begin code*⟩}{⟨*end code*⟩}{⟨*setup code*⟩}

Attaches codes to the ⟨*id*⟩-⟨*type*⟩-pair, where ⟨*id*⟩ must have been created using \pgfsys@new@id. The effect is that just before the id scope for this pair is created, the ⟨*setup code*⟩ is executed, then the scope is started, then the ⟨*begin code*⟩ is executed at the beginning, and, finally, ⟨*end code*⟩ gets executed just before the scope ends. Multiple calls of this macro accumulated.

120.14 Resource Description Framework Annotations (RDFa)

With certain output formats (in particular, with SVG) you can insert annotations into the output file following the standard set by the *resource description framework* (RDF), please consult the literature on RDF for an introduction to resource descriptions and ontologies.

The support for RDF annotations works as follows in PGF: You use the following commands before you create an id scope (using \pgfsys@begin@idscope). Then the attributes set by the commands will be added as an annotation to that object. Here is an example:

```
\pgfsys@rdf@resource{/fruits/apple}
\pgfsys@begin@idscope
  ...
\pgfsys@end@idscope
```

If SVG output is produced, this results in the following code in the SVG file:

```
<g resource="/fruits/apple">
  ...
</g>
```

Note that a call to `\pgfsys@begin@idscope` adds all the set attributes, but then clears the settings (globally). Thus, you should set all attributes more or less right before the id scope is created. For most of these command, if you call them multiple times before starting the id scope, the "last call wins", that is, later values overwrite earlier ones. In contrast, for the commands `\pgfsys@rdf@property`, `\pgfsys@rdf@rel`, `\pgfsys@rdf@rev`, as well as `\pgfsys@rdf@typeof`, the calls accumulate, that is, the texts passed in each call will all be added to the output, properly separated to form a list of values. Consider for instance:

```
\pgfsys@rdf@resource{/fruits/apple}
\pgfsys@rdf@resource{/fruits/watermelon}
\pgfsys@rdf@property{http://foo.com/props/juicy}
\pgfsys@rdf@property{http://foo.com/props/green}
\pgfsys@begin@idscope
  ...
\pgfsys@end@idscope
```

In the resulting id scope, we will have:

```
<g resource="/fruits/watermelon"
   property="http://foo.com/props/juicy http://foo.com/props/green">
  ...
</g>
```

`\pgfsys@rdf@about{`⟨*text*⟩`}`

Adds the RDF attribute `about="`⟨*text*⟩`"` to the next id scope (please see the RDFa specification for details on the semantics of `about` in the context of the resource description framework).

The following commands work the same way as the above command, except that the set attribute is different. Please see the RDFa specification for details on these attributes. Note that the `\pgfsys@rdf@inlist` command is the only one that takes no argument.

`\pgfsys@rdf@content{`⟨*text*⟩`}`

`\pgfsys@rdf@datatype{`⟨*text*⟩`}`

`\pgfsys@rdf@href{`⟨*text*⟩`}`

`\pgfsys@rdf@inlist`

`\pgfsys@rdf@prefix{`⟨*text*⟩`}`

`\pgfsys@rdf@property{`⟨*text*⟩`}`

`\pgfsys@rdf@rel{`⟨*text*⟩`}`

`\pgfsys@rdf@resource{`⟨*text*⟩`}`

`\pgfsys@rdf@rev{`⟨*text*⟩`}`

`\pgfsys@rdf@src{`⟨*text*⟩`}`

`\pgfsys@rdf@typeof{`⟨*text*⟩`}`

`\pgfsys@rdf@vocab{`⟨*text*⟩`}`

120.15 Reusable Objects System Commands

`\pgfsys@invoke{`⟨*literals*⟩`}`

This command gets protocolled literals and should insert them into the `.pdf` or `.dvi` file using an appropriate `\special`.

`\pgfsys@defobject{`⟨*name*⟩`}{`⟨*lower left*⟩`}{`⟨*upper right*⟩`}{`⟨*code*⟩`}`

Declares an object for later use. The idea is that the object can be precached in some way and then be rendered more quickly when used several times. For example, an arrow head might be defined and prerendered in this way.

The parameter ⟨*name*⟩ is the name for later use. ⟨*lower left*⟩ and ⟨*upper right*⟩ are PGF points specifying a bounding box for the object. ⟨*code*⟩ is the code for the object. The code should not be too fancy.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@useobject{⟨*name*⟩}{⟨*extra code*⟩}

Renders a previously declared object. The first parameter is the name of the object. The second parameter is extra code that should be executed right *before* the object is rendered. Typically, this will be some transformation code.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@marker@declare{⟨*macro*⟩}{⟨*code*⟩}

Declares a *marker* symbol for later use. The command is very similar to \pgfsys@defobject, but the use case is slightly different: The graphic object defined using the ⟨*code*⟩ is stored in such a way that it can be used as an *arrow tip marker symbol* in animations. The ⟨*macro*⟩ is set to an identifier by which the marker can be referenced later on.

This command has a default implementation and need not be implemented by a driver file.

\pgfsys@marker@use{⟨*macro*⟩}

Adds the marker object referenced by the ⟨*macro*⟩ to the current output.

This command has a default implementation and need not be implemented by a driver file.

120.16 Invisibility System Commands

All drawing or stroking or text rendering between calls of the following commands should be suppressed. A similar effect can be achieved by clipping against an empty region, but the following commands do not open a graphics scope and can be opened and closed "orthogonally" to other scopes.

\pgfsys@begininvisible

Between this command and the closing \pgfsys@endinvisible all output should be suppressed. Nothing should be drawn at all, which includes all paths, images and shadings. However, no groups (neither TEX groups nor graphic state groups) should be opened by this command.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 122.

\pgfsys@endinvisible

Ends the invisibility section, unless invisibility blocks have been nested. In this case, only the "last" one restores visibility.

This command has a default implementation and need not be implemented by a driver file.

This command is protocolled, see Section 122.

120.17 Page Size Commands

The following commands can be used to set the page size of a document in a "portable" way. Note, however, that many packages also (try to) set the page size.

These commands are typically not given inside a {pgfpicture}, but on the outer level of compilation.

\pgfsys@papersize{⟨*width*⟩}{⟨*height*⟩}

Inserts the necessary \specials for the current driver into the output stream to "locally" change the page size. Whether such a "local" change is possible depends strongly on the driver. For instance, dvips will honor the first call to this command that is part of the shipped-out document and will ignore all other uses. In contrast, pdftex will use the current value of the paper size for each page and, additionally, setting the papersize is local to the current TEX group.

\pgfsys@global@papersize{⟨*width*⟩}{⟨*height*⟩}

Like the previous command, only for drivers where setting the paper size parameters is a TEX-group-local operation, \global is prefixed to the setting of the page sizes.

\pgfsys@thepageheight

This macro expands to the current page's height, provided LATEX is used, otherwise a best guess is returned (currently just \the\vsize).

\pgfsys@thepagewidth

As above.

120.18 Position Tracking Commands

The following commands are used to determine the position of text on a page. This is a rather complicated process in general since at the moment when the text is read by TEX, the final position cannot be determined, yet. For example, the text might be put in a box which is later put in the headline or perhaps in the footline or perhaps even on a different page.

For these reasons, position tracking is typically a two-stage process. In a first stage you indicate that a certain position is of interest by *marking* it. This will (depending on the details of the backend driver) cause page coordinates or this position to be written to an `.aux` file when the page is shipped. Possibly, the position might also be determined at an even later stage. Then, on a second run of TEX, the position is read from the `.aux` file and can be used.

\pgfsys@markposition{⟨*name*⟩}

Marks a position on the page. This command should be given while normal typesetting is done such as in

```
The value of $x$ is \pgfsys@markposition{here}important.
```

It causes the position `here` to be saved when the page is shipped out.

\pgfsys@getposition{⟨*name*⟩}{⟨*macro*⟩}

This command retrieves a position that has been marked on an earlier run of TEX on the current file. The ⟨*macro*⟩ must be a macro name such as `\mymacro`. It will be redefined such that it is

- either just `\relax` or
- a `\pgfpoint...` command.

The first case will happen when the position has not been marked at all or when the file is typeset for the first time, when the coordinates are not yet available.

In the second case, executing ⟨*macro*⟩ yields the position on the page that is to be interpreted as follows: A coordinate like `\pgfpoint{2cm}{3cm}` means "2cm to the right and 3cm up from the origin of the page". The position of the origin of the page is not guaranteed to be at the lower left corner, it is only guaranteed that all pictures on a page use the same origin.

To determine the lower left corner of a page, you can call `\pgfsys@getposition` with ⟨*name*⟩ set to the special name `pgfpageorigin`. By shifting all positions by the amount returned by this call you can position things absolutely on a page.

Example: Referencing a point of the page:

```
The value of $x$ is \pgfsys@markposition{here}important.

Lots of text.

\hbox{\pgfsys@markposition{myorigin}%
\begin{pgfpicture}
  % Switch of size protocol
  \pgfpathmoveto{\pgfpointorigin}
  \pgfusepath{use as bounding box}

  \pgfsys@getposition{here}{\hereposition}
  \pgfsys@getposition{myorigin}{\thispictureposition}

  \pgftransformshift{\pgfpointscale{-1}{\thispictureposition}}
  \pgftransformshift{\hereposition}

  \pgfpathcircle{\pgfpointorigin}{1cm}
  \pgfusepath{draw}
\end{pgfpicture}}
```

120.19 Internal Conversion Commands

The system commands take TEX dimensions as input, but the dimensions that have to be inserted into PDF and PostScript files need to be dimensionless values that are interpreted as multiples of $\frac{1}{72}$in. For example, the TEX dimension 2*bp* should be inserted as `2` into a PDF file and the TEX dimension 10pt as `9.9626401`. To make this conversion easier, the following command may be useful:

`\pgf@sys@bp{`⟨*dimension*⟩`}`

Inserts how many multiples of $\frac{1}{72}$in the ⟨*dimension*⟩ is into the current protocol stream (buffered).

Example: `\pgf@sys@bp{\pgf@x}` or `\pgf@sys@bp{1cm}`.

Note that this command is *not* a system command that can/needs to be overwritten by a driver.

121 The Soft Path Subsystem

This section describes a set of commands for creating *soft paths* as opposed to the commands of the previous section, which created *hard paths*. A soft path is a path that can still be "changed" or "molded". Once you (or the PGF system) is satisfied with a soft path, it is turned into a hard path, which can be inserted into the resulting `.pdf` or `.ps` file.

Note that the commands described in this section are "high-level" in the sense that they are not implemented in driver files, but rather directly by the PGF-system layer. For this reason, the commands for creating soft paths do not start with `\pgfsys@`, but rather with `\pgfsyssoftpath@`. On the other hand, as a user you will never use these commands directly, they are described as part of the low-level interface.

121.1 Path Creation Process

When the user writes a command like `\draw (0bp,0bp) -- (10bp,0bp);` quite a lot happens behind the scenes:

1. The frontend command is translated by Ti*k*Z into commands of the basic layer. In essence, the command is translated to something like

   ```
   \pgfpathmoveto{\pgfpoint{0bp}{0bp}}
   \pgfpathlineto{\pgfpoint{10bp}{0bp}}
   \pgfusepath{stroke}
   ```

2. The `\pgfpathxxxx` commands do *not* directly call "hard" commands like `\pgfsys@xxxx`. Instead, the command `\pgfpathmoveto` invokes a special command called `\pgfsyssoftpath@moveto` and `\pgfpathlineto` invokes `\pgfsyssoftpath@lineto`.

   The `\pgfsyssoftpath@xxxx` commands, which are described below, construct a soft path. Each time such a command is used, special tokens are added to the end of an internal macro that stores the soft path currently being constructed.

3. When the `\pgfusepath` is encountered, the soft path stored in the internal macro is "invoked". Only now does a special macro iterate over the soft path. For each line-to or move-to operation on this path it calls an appropriate `\pgfsys@moveto` or `\pgfsys@lineto` in order to, finally, create the desired hard path, namely, the string of literals in the `.pdf` or `.ps` file.

4. After the path has been invoked, `\pgfsys@stroke` is called to insert the literal for stroking the path.

Why such a complicated process? Why not have `\pgfpathlineto` directly call `\pgfsys@lineto` and be done with it? There are two reasons:

1. The PDF specification requires that a path is not interrupted by any non-path-construction commands. Thus, the following code will result in a corrupted `.pdf`:

   ```
   \pgfsys@moveto{0}{0}
   \pgfsys@setlinewidth{1}
   \pgfsys@lineto{10}{0}
   \pgfsys@stroke
   ```

   Such corrupt code is *tolerated* by most viewers, but not always. It is much better to create only (reasonably) legal code.

2. A soft path can still be changed, while a hard path is fixed. For example, one can still change the starting and end points of a soft path or do optimizations on it. Such transformations are not possible on hard paths.

121.2 Starting and Ending a Soft Path

No special action must be taken in order to start the creation of a soft path. Rather, each time a command like `\pgfsyssoftpath@lineto` is called, a special token is added to the (global) current soft path being constructed.

However, you can access and change the current soft path. In this way, it is possible to store a soft path, to manipulate it, or to invoke it.

\pgfsyssoftpath@getcurrentpath{⟨*macro name*⟩}

This command will store the current soft path in ⟨*macro name*⟩.

\pgfsyssoftpath@setcurrentpath{⟨*macro name*⟩}

This command will set the current soft path to be the path stored in ⟨*macro name*⟩. This macro should store a path that has previously been extracted using the `\pgfsyssoftpath@getcurrentpath` command and has possibly been modified subsequently.

\pgfsyssoftpath@invokecurrentpath

This command will turn the current soft path in a "hard" path. To do so, it iterates over the soft path and calls an appropriate `\pgfsys@xxxx` command for each element of the path. Note that the current soft path is *not changed* by this command. Thus, in order to start a new soft path after the old one has been invoked and is no longer needed, you need to set the current soft path to be empty. This may seem strange, but it is often useful to immediately use the last soft path again.

\pgfsyssoftpath@flushcurrentpath

This command will invoke the current soft path and then set it to be empty.

121.3 Soft Path Creation Commands

\pgfsyssoftpath@moveto{⟨x⟩}{⟨y⟩}

This command appends a "move-to" segment to the current soft path. The coordinates ⟨x⟩ and ⟨y⟩ are given as normal TEX dimensions.

Example: One way to draw a line:

```
\pgfsyssoftpath@moveto{0pt}{0pt}
\pgfsyssoftpath@lineto{10pt}{10pt}
\pgfsyssoftpath@flushcurrentpath
\pgfsys@stroke
```

\pgfsyssoftpath@lineto{⟨x⟩}{⟨y⟩}

Appends a "line-to" segment to the current soft path.

\pgfsyssoftpath@curveto{⟨a⟩}{⟨b⟩}{⟨c⟩}{⟨d⟩}{⟨x⟩}{⟨y⟩}

Appends a "curve-to" segment to the current soft path with controls (a, b) and (c, d).

\pgfsyssoftpath@rect{⟨*lower left x*⟩}{⟨*lower left y*⟩}{⟨*width*⟩}{⟨*height*⟩}

Appends a rectangle segment to the current soft path.

\pgfsyssoftpath@closepath

Appends a "close-path" segment to the current soft path.

121.4 The Soft Path Data Structure

A soft path is stored in a standardized way, which makes it possible to modify it before it becomes "hard". Basically, a soft path is a long sequence of triples. Each triple starts with a *token* that identifies what is going on. This token is followed by two dimensions in braces. For example, the following is a soft path that means "the path starts at (0bp, 0bp) and then continues in a straight line to (10bp, 0bp)".

```
\pgfsyssoftpath@movetotoken{0bp}{0bp}\pgfsyssoftpath@linetotoken{10bp}{0bp}
```

A curve-to is hard to express in this way since we need six numbers to express it, not two. For this reasons, a curve-to is expressed using three triples as follows: The command

```
\pgfsyssoftpath@curveto{1bp}{2bp}{3bp}{4bp}{5bp}{6bp}
```

results in the following three triples:

```
\pgfsyssoftpath@curvetosupportatoken{1bp}{2bp}
\pgfsyssoftpath@curvetosupportbtoken{3bp}{4bp}
\pgfsyssoftpath@curvetotoken{5bp}{6bp}
```

These three triples must always "remain together". Thus, a lonely `supportbtoken` is forbidden.

In details, the following tokens exist:

- \pgfsyssoftpath@movetotoken indicates a move-to operation. The two following numbers indicate the position to which the current point should be moved.
- \pgfsyssoftpath@linetotoken indicates a line-to operation.
- \pgfsyssoftpath@curvetosupportatoken indicates the first control point of a curve-to operation. The triple must be followed by a \pgfsyssoftpath@curvetosupportbtoken.
- \pgfsyssoftpath@curvetosupportbtoken indicates the second control point of a curve-to operation. The triple must be followed by a \pgfsyssoftpath@curvetotoken.
- \pgfsyssoftpath@curvetotoken indicates the target of a curve-to operation.
- \pgfsyssoftpath@rectcornertoken indicates the corner of a rectangle on the soft path. The triple must be followed by a \pgfsyssoftpath@rectsizetoken.
- \pgfsyssoftpath@rectsizetoken indicates the size of a rectangle on the soft path.
- \pgfsyssoftpath@closepath indicates that the subpath begun with the last move-to operation should be closed. The parameter numbers are currently not important, but if set to anything different from {0pt}{0pt}, they should be set to the coordinate of the original move-to operation to which the path "returns" now.

122 The Protocol Subsystem

This section describes commands for *protocolling* literal text created by PGF. The idea is that some literal text, like the string of commands used to draw an arrow head, will be used over and over again in a picture. It is then much more efficient to compute the necessary literal text just once and to quickly insert it "in a single sweep".

When protocolling is "switched on", there is a "current protocol" to which literal text gets appended. Once all commands that needed to be protocolled have been issued, the protocol can be obtained and stored using `\pgfsysprotocol@getcurrentprotocol`. At any point, the current protocol can be changed using a corresponding setting command. Finally, `\pgfsysprotocol@invokecurrentprotocol` is used to insert the protocolled commands into the `.pdf` or `.dvi` file.

Only those `\pgfsys@` commands can be protocolled that use the command `\pgfsysprotocol@literal` internally. For example, the definition of `\pgfsys@moveto` in `pgfsys-common-pdf.def` is

```
\def\pgfsys@moveto#1#2{\pgfsysprotocol@literal{#1 #2 m}}
```

All "normal" system-level commands can be protocolled. However, commands for creating or invoking shadings, images, or whole pictures require special `\special`'s and cannot be protocolled.

`\pgfsysprotocol@literalbuffered{`⟨*literal text*⟩`}`

Adds the ⟨*literal text*⟩ to the current protocol, after it has been "`\edef`ed". This command will always be protocolled.

`\pgfsysprotocol@literal{`⟨*literal text*⟩`}`

First calls `\pgfsysprotocol@literalbuffered` on ⟨*literal text*⟩. Then, if protocolling is currently switched off, the ⟨*literal text*⟩ is passed on to `\pgfsys@invoke`.

`\pgfsysprotocol@bufferedtrue`

Turns on protocolling. All subsequent calls of `\pgfsysprotocol@literal` will append their argument to the current protocol.

`\pgfsysprotocol@bufferedfalse`

Turns off protocolling. Subsequent calls of `\pgfsysprotocol@literal` directly insert their argument into the current `.pdf` or `.ps`.

Note that if the current protocol is not empty when protocolling is switched off, the next call to `\pgfsysprotocol@literal` will first flush the current protocol, that is, insert it into the file.

`\pgfsysprotocol@getcurrentprotocol{`⟨*macro name*⟩`}`

Stores the current protocol in ⟨*macro name*⟩ for later use.

`\pgfsysprotocol@setcurrentprotocol{`⟨*macro name*⟩`}`

Sets the current protocol to ⟨*macro name*⟩.

`\pgfsysprotocol@invokecurrentprotocol`

Inserts the text stored in the current protocol into the `.pdf` or `.dvi` file. This does *not* change the current protocol.

`\pgfsysprotocol@flushcurrentprotocol`

First inserts the current protocol, then sets the current protocol to the empty string.

123 Animation System Layer

In conjunction with the right output format (namely SVG), you can specify that certain parts of you graphics can be animated. For this, there are a number of commands that cover, currently, what SVG 1.1 can do regarding animations. For a detailed introduction to animations, please see Section 116; the current section assumes that you are familiar with the concepts explained there.

The animation system consists of two layer itself: Commands starting with `\pgfsys@anim...` and commands starting with `\pgfsysanim`. These work as follows:

1. The commands starting with `\pgfsys@anim...` insert the actual animation commands into the output stream. A driver must implement these commands.

2. The command starting with `\pgfsysanim...` provide an. These commands, which are the ones that should be called by higher layers, implement the snapshot mechanism: When the command `\pgfsysanimsnapshot` is used, the `\pgfsysanim...` commands do *not* call the `\pgfsys@anim...` commands but, instead, insert non-animation commands to show the values of the attributes at the snapshot's time. To use this abstraction layer, you have to load the file `pgfsysanimations.code.tex`, which is not loaded by default (but is loaded by the PGF module `animations`).

The bottom line is that if you wish to implement a new driver, you need to implement the `\pgfsys@anim...` commands, if you use the animation layer, you call the `\pgfsysanim...` commands.

123.1 Animations and Snapshots

To add an animation to a graphic, use the following command (as described above, the first command is the one you actually call, the second is the one a driver implements):

`\pgfsysanimate{`⟨*attribute*⟩`}`

`\pgfsys@animate{`⟨*attribute*⟩`}`

The system layer animation subsystem follows the following philosophy: An animation always concerns an *attribute* of a *graphic object*. A *timeline* specifies how the attribute changes its value over time. Finally, a set of *keys* configures the animation as a whole like whether the timeline repeats or a event that triggers the start of the animation. The four parts of an animation, namely the *attribute*, the *graphic object*, the *timeline*, and the *keys*, are specified in different ways:

1. You choose the *attribute* using the system layer command `\pgfsysanimate`.
2. The *graphic object* whose attribute is to be animated is *always* specified by naming the ID of the graphic object *before* this object is created, see Section 120.13. (However, in the context of Ti*k*Z, it suffices that the animation is given in the object's options since these are executed before the actual object is created).
3. The *timeline* is specified using the commands `\pgfsysanimkeytime`, which specifies a time in seconds, and `\pgfsys@animation@val...`, which specify a value at this particular time. The timeline specifies for a sequence of times the values the attribute will have at these times. In between these *key times,* the value is interpolated.
4. The *animation keys* are specified by commands starting `\pgfsys@animation@...` and have the following effect: They set some property (like, say, whether the animation repeats or whether its effect is additive) to a given value *for the current TEX scope,* but do not create any animations. Rather, when `\pgfsysanimate` is called, a snapshot of the current values of all animation keys is taken and added to this animation of the attribute.

   When you set an animation key to a value, this will replace the value previously stored for the key (all keys are empty by default at the beginning).

   Note that animation keys are local to TEX scopes, not graphics scopes; indeed, they have little to do with the settings of the graphics scope other than the fact that a graphic scope is also a TEX scope and thereby influence the values of these keys.

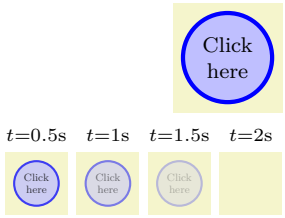
A typical example of how all of this works is the following:

```
\pgfsysanimkeyrepeatindefinite % Both of the following animations
                               % repeat indefinitely
{
  \pgfsysanimkeywhom{\someid}{}% The id of a later object
  \pgfsysanimkeyevent{}{}{click}{0}{begin}% Begin on a click ...
  \pgfsysanimkeytime{5}{1}{1}{0}{0} % Timeline starts after 5s
  \pgfsysanimvalscalar{0} % With a value of 0
  \pgfsysanimkeytime{8}{1}{1}{0}{0} % Timeline ends after 8s
  \pgfsysanimvalscalar{0.9} % With a value of 0.9
  \pgfsysanimate{fillopacity}% ... and the attribute is the fill opacity
}
{
  \pgfsysanimkeywhom{\someid}{}% The id of a later object
  \pgfsysanimkeyoffset{0}{begin}% Begin right away ...
  \pgfsysanimkeytime{1}{1}{1}{0}{0} % Timeline starts after 1s
  \pgfsysanimvalcurrent % With the current value
  \pgfsysanimkeytime{5}{1}{1}{0}{0} % Timeline ends after 5s
  \pgfsysanimvaldimension{5pt} % With a value of 5pt
  \pgfsysanimate{linewidth}% ... and the attribute is the line width
}
```

As a real-life example, consider the following definitions, which will be used in many examples in the rest of this section: Both take three parameters: The PGF/Ti*k*Z name of a to-be animated object, a type (relevant for objects that have subtypes or parts), and some code for triggering the actual animation. The animation will always start when the button is clicked. The second macro sets up things in such a way that the animation will last two seconds, while the first leaves the timing open.

```
\def\animationexample#1#2#3{
  \tikz[fill=blue!25, draw=blue, ultra thick] {
    \pgfidrefnextuse{\objid}{#1}
    \pgfsysanimkeywhom{\objid}{#2}
    \pgfidrefnextuse{\nodeid}{node}
    \pgfsysanimkeyevent{\nodeid}{}{click}{}{begin}
    #3
    \node [font=\scriptsize, circle, fill, draw, align=center]
      (node) {Click \\ here};
  }
}
```

Now the example, where the circle will disappear, when clicked:




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}
  \pgfsysanimate{opacity}
}
```

The "opposite" of `\pgfsysanimate` is the following command:

`\pgfsysanimsnapshot{`⟨*time*⟩`}`

Use this command in a scope prior to calling any other commands documented in this section concerning the configuration of animations. In this case, all uses of `\pgfsysanimate` inside the TEX scope no longer insert an animation into the output file. Instead, a "snapshot" is inserted of what the animation "would like at time ⟨*time*⟩". For instance, if an animation inserts a movement of an object by 4cm over a time of 2s and you take a snapshot with ⟨*time*⟩ = 2s, you get a picture in which the object is moved by 1cm.

A lot of care has been taken to make the output produced by the snapshot be as close as possible as what the animation really would look like at time ⟨*time*⟩, but note the following restrictions:

1. Interactive events of all kinds (like `click` or `mouseover`) make little sense for snapshots, which are created once and for all during the typesetting of the document. For this reason, all events are ignored for snapshots (even sync bases, and `begin` and `end` events, which might make some sense also in a snapshot setting).

   However, there is one command which helps you with "simulating" the effect of events:

\pgfsysanimkeysnapshotstart{⟨*time offset*⟩}

This command specifies that for the current animation the "moment `0s`" of the timeline is at ⟨*time offset*⟩. Thus, it works like `\pgfsysanimkeyoffset`, only the offset is now solely for the snapshot timeline. It has no effect on the actual animation.

2. The command `\pgfsysanimvalcurrent` cannot be used with snapshots since PGF has no chance of computing the correct current value. You always have to specify the start value explicitly.
3. The computation of time splines (entry and exit splines) and the accumulation of values after a large number of repeats may not be numerically stable.




```
\usetikzlibrary {animations}
\foreach \t in {0.5,1,1.5,2} {
  \pgfsysanimsnapshot{\t}
  \tikz {
    \pgfidrefnextuse{\objid}{node}
    \pgfsysanimkeywhom{\objid}{}
    \pgfsysanimkeytime{0}{1}{1}{0}{0}
    \pgfsysanimvalscalar{1}
    \pgfsysanimkeytime{2}{1}{1}{0}{0}
    \pgfsysanimvalscalar{0}
    \pgfsysanimate{opacity}
    \node (node) [draw = blue, very thick, fill=blue!20, circle] {Hi};
  }
}
```

\pgfsysanimsnapshotafter{⟨*time*⟩}

Works like the previous command, only the "moment" that ⟨*time*⟩ refers to is conceptually ⟨*time*⟩ + ϵ: When timeline specifies several values for ⟨*time*⟩, this command will select the last value at ⟨*time*⟩, while `\pgfsnapshot` will select the first value at ⟨*time*⟩. Similarly, when a timeline ends at ⟨*time*⟩, `\pgfsnapshot` will select the last value of the timeline while `\pgfsnapshotafter` will not apply the animation any more.

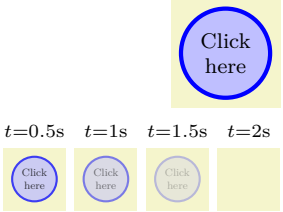
123.2 Commands for Animating an Attribute: Color, Opacity, Visibility, Staging

The commands from this and the next sections specify that some attribute should be animated. We start with rather basic animation attributes for color, visibility, and opacity.

\pgfsysanimate{opacity}

Adds an animation of the opacity to the graphic object specified using `\pgfsysanimkeywhom`. If the driver supports this, this is a bit different from animating the fill and stroke opacities individually: Paths are treated as transparency groups for this key. Typically, "this is what you want".

Specify values with `\pgfsysanimvalscalar`.

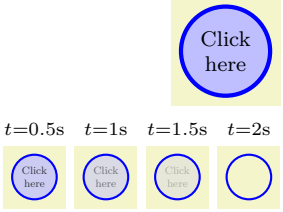



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}
  \pgfsysanimate{opacity}
}
```

\pgfsysanimate{fillopacity}

Adds an animation of only the opacity of fill operations.

Specify values with `\pgfsysanimvalscalar`.

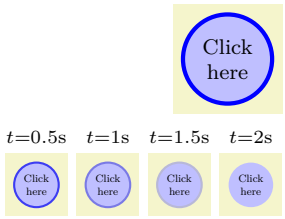



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}
  \pgfsysanimate{fillopacity}
}
```

`\pgfsysanimate{strokeopacity}`

Adds an animation of only the opacity of draw (stroke) operations.

Specify values with `\pgfsysanimvalscalar`.

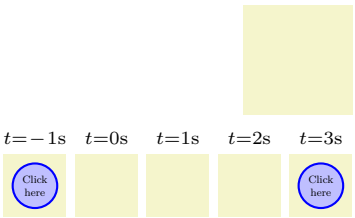



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}
  \pgfsysanimate{strokeopacity}
}
```

`\pgfsysanimate{visibility}`

Adds an animation of the "visibility".

Specify values with `\pgfsysanimvaltext`. However, only two values are allowed: `visible` and `hidden`.

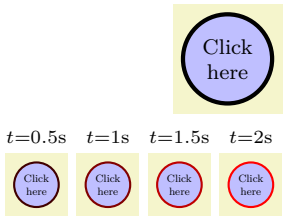



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltext{hidden}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltext{hidden}
  \pgfsysanimate{visibility}
}
```

`\pgfsysanimate{strokecolor}`

Adds an animation of the stroke color.

Specify values with `\pgfsysanimvalcolorrgb` and friends.

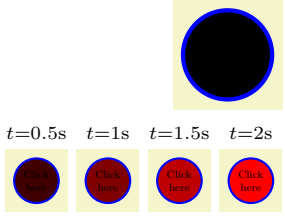



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalcolorrgb{0}{0}{0}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalcolorrgb{1}{0}{0}
  \pgfsysanimate{strokecolor}
}
```

`\pgfsysanimate{fillcolor}`

Adds an animation of the fill color.

Specify values with `\pgfsysanimvalcolorrgb` and friends.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalcolorrgb{0}{0}{0}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalcolorrgb{1}{0}{0}
  \pgfsysanimate{fillcolor}
}
```

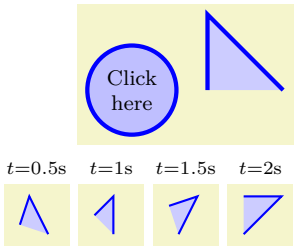
123.3 Commands for Animating an Attribute: Paths and Their Rendering

The following attributes influence paths and how they are rendered.

`\pgfsysanimate{path}`

Adds an animation of the path itself. That means that the path will morph its form from one path to another. When morphing a path, all "values", which are the paths, must consist of the *exact same* path construction commands; they may only differ with respect to the numbers used in these descriptions.

Specify values with `\pgfsysanimvalpath`.




```
\usetikzlibrary {animations}
\animationexample{my path}{path}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{0cm}%
                     \pgfsys@lineto{1cm}{1cm}%
                     \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{1cm}%
                     \pgfsys@lineto{2cm}{1cm}%
                     \pgfsys@lineto{1cm}{0cm}}
  \pgfsysanimate{path}
  \filldraw [ultra thick,draw=blue,fill=blue!20, name=my path]
    (1,0) -- (1,1) -- (2,0); }
```

You can attach arrow tips to paths that are animated and these arrow tips will correctly "rotate and move along" with the path's end points *if* you take the following points into considerations:

- Arrow tips that "rotate and move along" with a path must be specified using a special animation command, see below. The normal arrow tips added to a path would *not* be animated automatically and, indeed, if you add arrow tips to a path using `\pgfsetarrows` and then animate the path, you will get an error message.
- Internally, the arrow tips that "rotate and move along" are drawn using so-called *markers*. These are little graphic objects that can be added to the start and end of paths and that are automatically rotated and move along with the path.

  In principle, the rendering rules used by SVG for markers are the same as for normal arrow tips: The markers are rotated and moved so that the point along a tangent of the path at the start or end of the path. However, when it comes to special cases such as a path with multiple segments, a path with degenerate segments, a closed path, and so on, the rules used by for instance SVG may differ from the placement that PGF will compute.

  Thus, it is best to add arrow tips only to "normal" paths consisting of a single open path segment whose ends can be shortened a bit without causing degeneration.
- When an arrow tip is added to a path, the path must typically be shortened a bit so that the *tip* of the arrow ends where the path would usually end. This shortening is not done by the system layer for to-be-animated paths; you must compute and then animate these shortened paths yourself. (However, the basic layer animation module will do this for you, so you only have to worry about this when you use the system layer directly.)

Let us now have a look at how we add arrow tip markers:

`\pgfsysanimkeytipmarkers{`⟨*start marker*⟩`}{`⟨*end marker*⟩`}`

`\pgfsys@animation@tip@markers{`⟨*start marker*⟩`}{`⟨*end marker*⟩`}`

This command specifies that during a path animation the two markers provided as parameters should be added (and rotated and moved along with the path) at the start and end. The ⟨*start marker*⟩ must either be empty (in which case no marker is added at the start) or it must be a macro storing a value returned by the command `\pgfsys@marker@declare`. In this case, the marker declared symbol will be added to the start during the animation. The same situation applies to the end of the path.

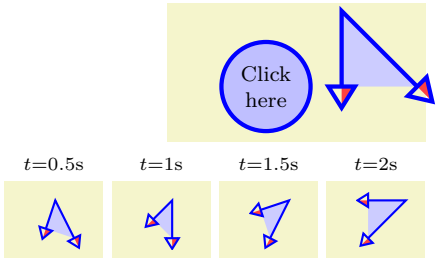
As pointed out earlier, only arrow tips / markers added to paths using this command will be animated along with the path. In particular, you should *not* add arrow tips to to-be-animated paths using `\pgfsetarrow`. However, when you use a base value (`\pgfsys@animation@base`) to set a path, the arrow tips will also be added to this base path.

To sum up, the "correct" way of adding arrow tips to a path that is animated is to proceed as follows:

1. You specify arrow tips for a path using this command.
2. You specify times and values of the to-be-animated path, shortened as necessary to accommodate the length of the arrow tips.
3. You specify the first (or, possibly, some other) value in the time–value sequence as a base value.
4. You create a path animation that applies to a future path.

5. You create this future path as an empty path without arrow tips and draw it. Because of the setting of the base value, instead of the empty path the base path will be used as the “real” path and the animation’s arrow tips will be added as arrow tips.

When you have more than one animation for a given path, these different animations may use different arrow tips / markers. This allows you to animate (change) which arrow tip is used on a path over time.

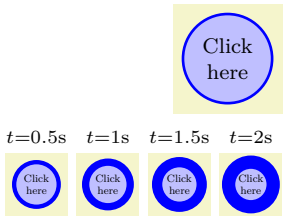



```
\usetikzlibrary {animations}
% Declare a marker:
\pgfsys@marker@declare\mymarker{%
  \pgfscope%
    \pgfsetcolor{red!75}%
    \pgfpathmoveto{\pgfpoint{0pt}{5pt}}\pgfpathlineto{\pgfpoint{8pt}{0pt}}%
    \pgfpathlineto{\pgfpoint{0pt}{0pt}}\pgfpathclose%
    \pgfusepathqfill%
  \endpgfscope%
  \pgfpathmoveto{\pgfpoint{0pt}{5pt}}\pgfpathlineto{\pgfpoint{8pt}{0pt}}%
  \pgfpathlineto{\pgfpoint{0pt}{-5pt}}\pgfpathclose%
  \pgfusepathqstroke%
}%
\animationexample{my path}{path}{
  \pgfsysanimkeytipmarkers{\mymarker}{\mymarker}
  \pgfsysanimkeybase
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{0cm}%
                     \pgfsys@lineto{1cm}{1cm}%
                     \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{0cm}%
                     \pgfsys@lineto{1cm}{1cm}%
                     \pgfsys@lineto{2cm}{0cm}}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalpath{\pgfsys@moveto{1cm}{1cm}%
                     \pgfsys@lineto{2cm}{1cm}%
                     \pgfsys@lineto{1cm}{0cm}}
  \pgfsysanimate{path}
  \filldraw [ultra thick,draw=blue,fill=blue!20, name=my path];
  \path (1,0) (2,1);}
```

`\pgfsysanimate{linewidth}`

Adds an animation of the line width.

Specify values with `\pgfsysanimvaldimension`.

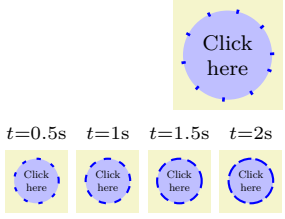



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldimension{1pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldimension{10pt}
  \pgfsysanimate{linewidth}
}
```

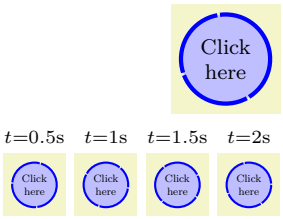
`\pgfsysanimate{dash}`

Adds an animation of the dash phase and pattern (like `\pgfsys@setdash`).

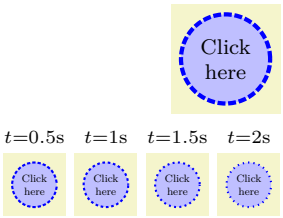
Specify values with `\pgfsysanimvaldash`.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldash{1pt,10pt}{0pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldash{10pt,3pt}{0pt}
  \pgfsysanimate{dash}
}
```




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldash{1cm,1pt}{0pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldash{1cm,1pt}{1cm}
  \pgfsysanimate{dash}
}
```




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaldash{3pt,1pt}{0pt}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaldash{1pt,3pt}{10pt}
  \pgfsysanimate{dash}
}
```

123.4 Commands for Animating an Attribute: Transformations and Views

The commands in this section allow you to animate the canvas transformation matrix of a scope. However, there is one command that needs to be explained first.

\pgfsysanimkeycanvastransform{⟨*pre*⟩}{⟨*post*⟩}

\pgfsys@animation@canvas@transform{⟨*pre*⟩}{⟨*post*⟩}

In order to animate the canvas, you specify that, for instance, the canvas should be shifted over, say, one second by 2cm from left to right. In order to specify this, you specify that an additional shift should be added to the canvas transformation matrix that starts out as $(0, 0)$ and ends at $(2\,\text{cm}, 0)$. However, it is not immediately clear what "to the right" or $(2\,\text{cm}, 0)$ actually means: "Right" relative to the paper? "Right" relative to the coordinate system at the point when the animation is created? "Right" relative to the object's local coordinate system?

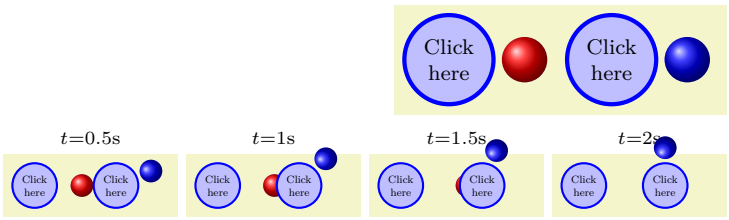
Using this command you can specify the coordinate system relative to which all canvas animations are specified. In detail, when you add an animation a of the canvas of an object foo, the following happens:

1. We start with the canvas transformation matrix that is installed when the object starts. More precisely, this is the canvas transformation matrix that is in force when the command `\pgfsys@begin@idscope` is called for the object. The canvas transformation matrix that is in force when the animation is created (which is typically "way before" the object is created and may even be in a totally different graphics scope) is irrelevant for the animation.
2. Now, when the object is created, the code ⟨*pre*⟩ is executed. It should call `\pgfsys@transformcm` at most once. This canvas transformation is added to the object's canvas transformation.
3. Now, the animation a of the canvas is relative to the resulting canvas transformation. That means, when the animation shifts the object "to the right" the animation will actually be along the current direction of "right" in the canvas transformation resulting from the two transformations above.
4. Finally, at the point of creation of the to-be-animation object the code ⟨*post*⟩ is executed. Again, the code should call `\pgfsys@transformcm` at most once. The resulting transformation is also added to the object's canvas transformation, but does *not* influence the animation.

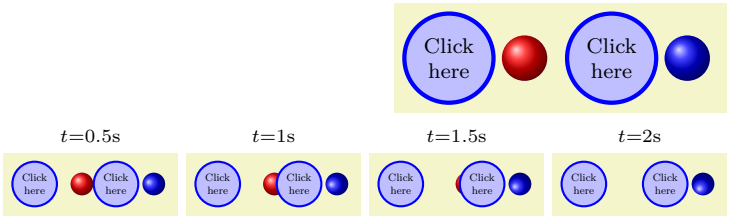
The net effect of the above is that, normally, you use the ⟨*pre*⟩ code to setup a transformation matrix relative to which you wish to perform your animation and, normally, you use ⟨*post*⟩ to undo this transformation (using the inverted matrix) to ensure that when no animation is in force, the object is placed at the same position as if no animation were used.

Let us now have a look at some examples. We use the following macro, which takes a pre and a post code and animates a red ball over 1cm to the right in two seconds and rotates the blue ball over 90° around the origin. The ball is placed at $(1, 0)$.

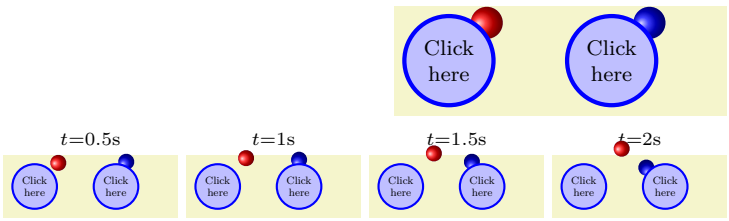
```
\def\animationcanvasexample#1#2{%
  \animationexample{ball}{}{%
    \pgfsysanimkeycanvastransform{#1}{#2}%
    \pgfsysanimkeytime{0}{1}{1}{0}{0}
    \pgfsysanimvaltranslate{0cm}{0cm}%
    \pgfsysanimkeytime{2}{1}{1}{0}{0}
    \pgfsysanimvaltranslate{1cm}{0cm}%
    \pgfsysanimate{translate}
    \fill [ball color=red,name=ball] (1,0) circle [radius=3mm]; }
  \animationexample{ball}{}{%
    \pgfsysanimkeycanvastransform{#1}{#2}%
    \pgfsysanimkeytime{0}{1}{1}{0}{0}
    \pgfsysanimvalscalar{0}%
    \pgfsysanimkeytime{2}{1}{1}{0}{0}
    \pgfsysanimvalscalar{90}%
    \pgfsysanimate{rotate}
    \fill [ball color=blue,name=ball] (1,0) circle [radius=3mm]; } }
```



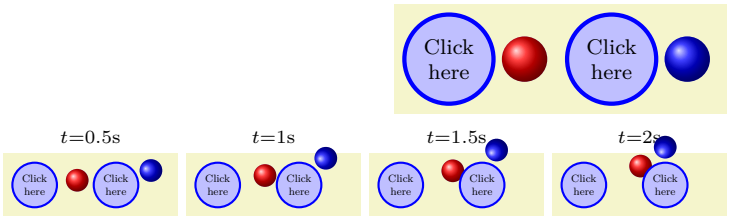

```
\usetikzlibrary {animations}
\animationcanvasexample
{}
{}
```




```
\usetikzlibrary {animations}
\animationcanvasexample
{\pgfsys@transformshift{10mm}{0mm}}
{\pgfsys@transformshift{-10mm}{0mm}}
```




```
\usetikzlibrary {animations}
\animationcanvasexample
{\pgfsys@transformcm{0.5}{0.5}{-0.5}{0.5}
                    {0pt}{0pt}}
{}
```

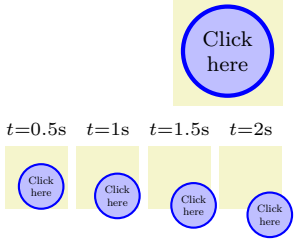



```
\usetikzlibrary {animations}
\animationcanvasexample
{\pgfsys@transformcm{0.5}{0.5}{-0.5}{0.5}
                    {0pt}{0pt}}
{\pgfsys@transformcm{1}{-1}{1}{1}
                    {0pt}{0pt}}
```

\pgfsysanimate{translate}

Adds an (additional) translate animation. Effectively, this causes the group to be shifted to different positions.

Specify values with \pgfsysanimvaltranslate.

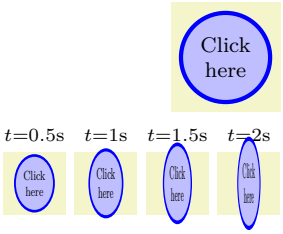



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{5mm}{-1cm}
  \pgfsysanimate{translate}
}
```

`\pgfsysanimate{scale}`

Adds an animation of the scaling relative to the origin. This causes a scaling of the canvas, including fonts and line widths.

Specify values with `\pgfsysanimvalscale`.

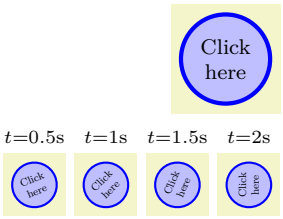



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscale{1}{1}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscale{0.5}{2}
  \pgfsysanimate{scale}
}
```

`\pgfsysanimate{rotate}`

Adds a rotation animation around the origin.

Specify values with `\pgfsysanimvalscalar`.

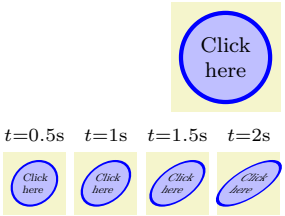



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{90}
  \pgfsysanimate{rotate}
}
```

`\pgfsysanimate{skewx}`

Adds an animation of a skewing of the canvas along the x-axis. Unlike the `slant` options of Ti*k*Z, the skew is given in degrees.

Specify values with `\pgfsysanimvalscalar`.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{45}
  \pgfsysanimate{skewx}
}
```

`\pgfsysanimate{skewy}`

Adds an animation of a skewing of the canvas along the y-axis.

Specify values with `\pgfsysanimvalscalar`.

`\pgfsysanimate{motion}`

Works a bit like `\pgfsysanimvaltranslate`: It also adds an animated shift transformation of the canvas. However, instead of specifying some shift coordinates as values, you now specify a whole path (which may include curves). The effect is that an animated translate transformation for the different points on this path gets installed. Furthermore, if you use `\pgfsysanimkeyrotatealong`, an additional adaptive rotation transformation will be added so that the animated graphic scope "points along" the path.

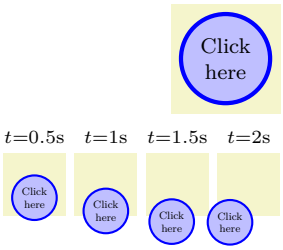
You specify the path along which you wish to move objects along using `\pgfsysanimkeymovealong`. You use the timeline to specify how far the object gets moved along this path using scalar values where `0` is the beginning of the path and `1` is the end. Thus, setting the timeline to the scalar value of `0` at time t_0 and to `1` at time t_1 will cause the object o move along the complete path between times t_0 and t_1.

Specify values with `\pgfsysanimvalscalar`.

`\pgfsysanimkeymovealong{`⟨*path*⟩`}`

\pgfsys@animation@movealong{⟨*path*⟩}

Defines the ⟨*path*⟩ along which the motion will occur. It will simply be executed and must call \pgfsys@lineto and similar path-construction commands, but should not call other commands.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeymovealong{
    \pgfsyssoftpath@movetotoken{0pt}{0pt}
    \pgfsyssoftpath@linetotoken{0pt}{-5mm}
    \pgfsyssoftpath@curvetosupportatoken{0pt}{-1cm}%
    \pgfsyssoftpath@curvetosupportbtoken{0pt}{-1cm}%
    \pgfsyssoftpath@curvetotoken{-5mm}{-1cm} }
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimate{motion}
}
```

\pgfsysanimkeynorotatealong

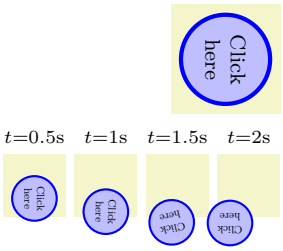
\pgfsys@animation@norotatealong

Indicates that no additional rotation should be added during the movement. This is the default.

\pgfsysanimkeyrotatealong

\pgfsys@animation@rotatealong

Indicates that the to-be-animated group should be rotated automatically so that it points along the path as time progresses. This option is only applicable to motion animations.

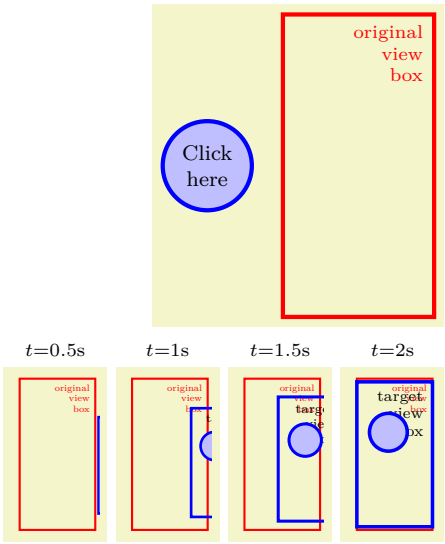



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeyrotatealong
  \pgfsysanimkeymovealong{%
    \pgfsyssoftpath@movetotoken{0pt}{0pt}%
    \pgfsyssoftpath@linetotoken{0pt}{-5mm}%
    \pgfsyssoftpath@curvetosupportatoken{0pt}{-1cm}%
    \pgfsyssoftpath@curvetosupportbtoken{0pt}{-1cm}%
    \pgfsyssoftpath@curvetotoken{-5mm}{-1cm}}
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalscalar{0}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalscalar{1}
  \pgfsysanimate{motion}
}
```

\pgfsysanimate{viewbox}

Adds an animation of the view box. The graphic scope to which this animation is added must have been created using \pgfsys@viewboxmeet or \pgfsys@viewboxslice; adding it to other scopes has no effect. Note that this command does *not* change or animate the scope's transformation matrix – it only animates the "what we see through the view box".

Specify values with \pgfsysanimvalviewbox.




```
\usetikzlibrary {animations,views}
\animationexample{my view}{view}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvalviewbox{-10mm}{-20mm}{10mm}{20mm}%
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvalviewbox{15mm}{-15mm}{27mm}{8mm}%
  \pgfsysanimate{viewbox}
  \scoped [xshift=2cm] {
    \draw [red] (-1,-2) rectangle (1,2) node [font=\scriptsize,
               below left, align=right] {original\\ view\\ box};
    \scoped [meet={(-1,-2) (1,2)}, name=my view]  {
      \draw (-5mm,-15mm) rectangle (7mm,8mm)
        node [font=\scriptsize, align=right, below left]
          {target\\ view\\ box};
      \filldraw (0,0) circle [radius=3mm];
} } }
```

123.5 Commands for Specifying the Target Object

\pgfsysanimkeywhom{⟨*id*⟩}{⟨*type*⟩}

\pgfsys@animation@whom{⟨*id*⟩}{⟨*type*⟩}

Sets the target of the animation. The {⟨*id*⟩} must previously have been created using \pgfsys@new@id, {⟨*type*⟩} must be a type (the empty type is also allowed). See Section 120.13 for details on ids and types.

123.6 Commands for Specifying Timelines: Specifying Times

Animations are specified using *timelines*, which are functions mapping times to values for these times. The functions are cubic splines, which are specified using time–value pairs plus control points.

In order to specify a time–value pair, you first use the command \pgfsysanimkeytime to specify a time. Next, you use \pgfsysanimval... to specify a value, which adds the time–value pair to the timeline. Note that the times must be given in non-decreasing order. Between time–value pairs, the values are interpolated using a spline.

The first and last times of the timeline are a bit special: The timeline starts on the first time and the duration of the timeline is the difference between the first and last time. "Starting" on the start time actually means that any beginnings (see the commands for specifying beginnings and endings) get as offset the start time; similarly end times are offset by this value.

\pgfsysanimkeytime{⟨*time*⟩}{⟨*entry spline control x*⟩}{⟨*entry spline control y*⟩}{⟨*exit spline control x*⟩}{⟨*exit spline control y*⟩}

\pgfsys@animation@time{⟨*time*⟩}{⟨*entry spline control x*⟩}{⟨*entry spline control y*⟩}{⟨*exit spline control x*⟩}{⟨*exit spline control y*⟩}

The ⟨*time*⟩ is a number representing seconds (so `0.5` means 500 ms).

The spline between a time–value pair and the next is specified using the four parameters following the time. The first two of these specify the second control point of the interval preceding the time–value pair (called the "entry" control point), the last two parameters specify the first control point of the interval following the pair (called the "exit" control point). Consider for instance, the following calls:

```
\pgfsysanimkeytime{10}{0.1}{0.2}{0.3}{0.4}
\pgfsysanimvalscalar{100}
\pgfsysanimkeytime{15}{0.5}{0.6}{0.7}{0.8}
\pgfsysanimvalscalar{200}
```

This will create (at least) the time interval [10 s, 15 s] and the control points for this interval will be (0.3, 0.4) and (0.5, 0.6).

Control points are specified in a different "coordinate" system from the time–value pairs themselves: While the time–value pairs are specified using a number representing seconds and a value using some

special commands, the control points are specified as numbers between 0 and 1, each time representing a fraction of the time interval or the value interval. In the example, the time interval is [10 s, 15 s] and the value interval is [100, 200]. This means that a control point of (0.3, 0.4) actually refers to the time–value (11.5 s, 140). The "time–value curve" in the interval thus "`(10s,100) .. controls (11.5s,140) and (12.5s,160) .. (15s,200)`".

Note that by setting the control points always to (1, 1) and (0, 0) you get a linear interpolation between time–value pairs.

Two special cases are the following: When the two last parameters, the exit spline, take the special values `stay` and `0`, the attribute's value "stays" until the next value for the next time (it then "jumps" to the next value then). This corresponds, roughly, to an "infinite" ⟨*exit spline control x*⟩. Similarly, when the entry spline parameters take the special values `jump` and `1`, the value immediately jumps from the previous value to the next value when the previous value was specified.

`\pgfsysanimkeybase`

`\pgfsys@animation@base`

This command can be used in any place where `\pgfsys@animation@time` is usually used. The effect is that the next value does not become part of the timeline, but will become the value used for the attribute when no animation is active. (Normally, when the animation is not active, no value is set at all and the value is inherited from the surrounding scope.)

It may happen that there is more than one timeline active that is "trying to modify" a given attribute. In this case, the following rules are used to determine, which timeline "wins":

1. If no animation is active at the current time (all animation either have not yet started or they have already ended), then the base value given in the animation encountered last in the code is used. (If there are no base values, the attribute is taken from the surrounding scope.)
2. If there are several active animations, the one that has started last is used and the its value is used.
3. If there are several active animations that have started at the same time, the one that comes last in the code is used.

Note that these rules do not apply to transformations of the canvas since these are always additive (or, phrased differently, they are always all active and the effects accumulate).

123.7 Commands for Specifying Timelines: Specifying Values

The following commands are used to specify the values of a timeline. Each use of one of the following commands adds one time–value pair to the timeline. Which of the commands must be used depends on the type of the to-be-animated attribute (see the `\pgfsysanimate` command instances, which list the command that must be used).

`\pgfsysanimvalcurrent`

`\pgfsys@animation@val@current`

Creates a time–value pairs where the value is the current value that the attribute has. This command can only be used in conjunction with "real" animations, when you use it with a snapshot an error is raised.

`\pgfsysanimvaltext{`⟨*text*⟩`}`

`\pgfsys@animation@val@text{`⟨*text*⟩`}`

Creates a time–value pairs where the value is some text. Which texts are permissible depends on the to-be-animated attribute.

`\pgfsysanimvalscalar{`⟨*number*⟩`}`

`\pgfsys@animation@val@scalar{`⟨*number*⟩`}`

Creates a time–value pairs where the value is a number like `0.5` or `-2.25`.

\pgfsysanimvaldimension{⟨*dimension*⟩}

\pgfsys@animation@val@dimension{⟨*dimension*⟩}

Creates a time–value pairs where the value is a TEX dimension like `0.5pt` or `-2in`.

\pgfsysanimvalcolorrgb{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

\pgfsys@animation@val@color@rgb{⟨*red*⟩}{⟨*green*⟩}{⟨*blue*⟩}

Creates a time–value pairs where the value is color specified by three fractional values between 0 and 1 for the red, the green, and the blue part.

\pgfsysanimvalcolorcmyk{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

\pgfsys@animation@val@color@cmyk{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}{⟨*black*⟩}

Creates a time–value pairs where the value is color specified by four fractional values between 0 and 1 for the cyan, magenta, yellow, and black part.

\pgfsysanimvalcolorcmy{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

\pgfsys@animation@val@color@cmy{⟨*cyan*⟩}{⟨*magenta*⟩}{⟨*yellow*⟩}

Like the `\pgfsysanimvalcolorcmyk` only without the black part.

\pgfsysanimvalcolorgray{⟨*gray value*⟩}

\pgfsys@animation@val@color@gray{⟨*gray value*⟩}

Creates a time–value pairs where the value is gray value (a fraction between 0 and 1).

\pgfsysanimvalpath{⟨*low-level path construction commands*⟩}

\pgfsys@animation@val@path{⟨*low-level path construction command*⟩}

Creates a time–value pairs where the value is path. The ⟨*low-level commands*⟩ must consist of a sequence of path construction commands like `\pgfsys@lineto` or `\pgfsyssoftpath@linetotoken` (more precisely, the commands must form a list of TEX tokens and dimensions surrounded by braces). For each call of this command, the sequence of tokens and numbers must be the some. During the animation, only and exactly the numbers will be interpolated.

\pgfsysanimvaltranslate{⟨*x dimension*⟩}{⟨*y dimension*⟩}

\pgfsys@animation@val@translate{⟨*x dimension*⟩}{⟨*y dimension*⟩}

Creates a time–value pairs where the value is a coordinate. The dimensions must be TEX dimensions.

\pgfsysanimvalscale{⟨*x scale*⟩}{⟨*y scale*⟩}

\pgfsys@animation@val@scale{⟨*x scale*⟩}{⟨*y scale*⟩}

Creates a time–value pairs where the value is pair of scalar values.

\pgfsysanimvalviewbox{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}

\pgfsys@animation@val@viewbox{⟨x_1⟩}{⟨y_1⟩}{⟨x_2⟩}{⟨y_2⟩}

Creates a time–value pairs where the value is view box. The lower left corner is given by (x_1, y_1), consisting of two TEX dimensions, and the upper right corner is (x_2, y_2).

\pgfsysanimvaldash{⟨*pattern*⟩}{⟨*phase*⟩}

\pgfsys@animation@val@dash{⟨*pattern*⟩}{⟨*phase*⟩}

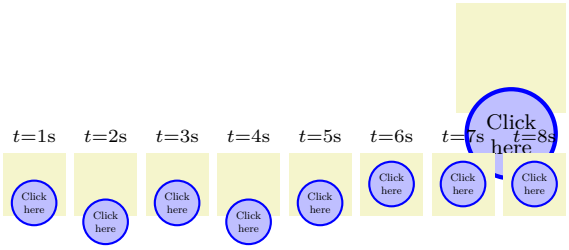
Creates a time–value pairs where the value is dash pattern and phase with the same syntax as `\pgfsys@setdash`.

123.8 Commands for Specifying Timing: Repeats

\pgfsysanimkeyrepeatnumber of times

\pgfsys@animation@repeat{⟨*number of times*⟩}

Specifies that the animation should repeat the specified ⟨*number of times*⟩, which may be a fractional number.

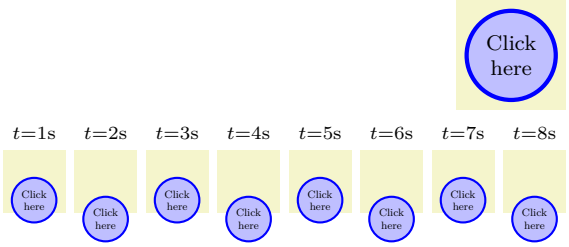



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrepeat{2.5}
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyrepeatindefinite

\pgfsys@animation@repeat@indefinite

Specifies that the animation should repeat indefinitely.

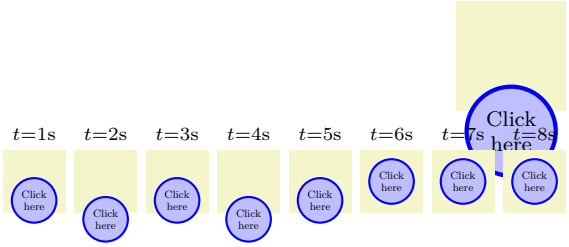



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrepeatindefinite
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyrepeatdur⟨*seconds*⟩

\pgfsys@animation@repeat@dur⟨*seconds*⟩

Specifies that the animation should repeat until ⟨*seconds*⟩ have elapsed.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrepeatdur{5}
  \pgfsysanimate{translate} }
```

123.9 Commands for Specifying Timing: Beginning and Ending

Normally, animations start when a graphic is displayed. Using the following commands, you can change this behavior: For instance, you can specify that the animation should start when, say, some button has been pressed or a key has been hit. Similarly, you can also use the commands to specify that the animation should stop early, for instance when a button is pressed.

Note that all of the commands for specifying a nonstandard begin (or end) of an animation's timeline refer to when the time 0 s of the timeline should actually be. If the first time–value point for a timeline is at, say, 2 s and you specify that the begin of the animation is one second after the click of a button, the attribute will attain the value specified by the time–value point three seconds after the button has been pressed.

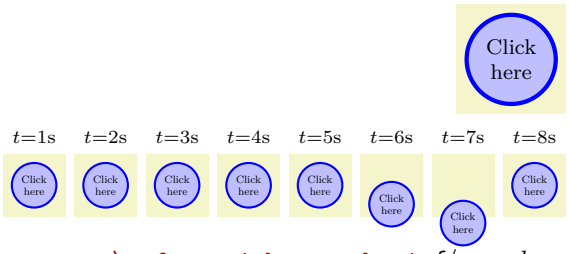
All of the following commands take either the text `begin` or `end` as their last argument.

You can call the commands several times. This will result in several different possible beginnings (or endings).

\pgfsysanimkeyoffset{⟨*time offset*⟩}{⟨*begin or end*⟩}

`\pgfsys@animation@offset{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

Specifies that (in addition to any other beginnings or endings) the animation's timeline should begin (or end) ⟨*time offset*⟩ many seconds after the graphic is shown. For instance, in the next example the animation will start automatically after 5 s *or* when then button is pressed.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyoffset{5}{begin}
  \pgfsysanimate{translate} }
```

`\pgfsysanimkeysyncbegin{`⟨*sync base id*⟩`}{`⟨*type*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

`\pgfsys@animation@syncbegin{`⟨*sync base id*⟩`}{`⟨*type*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

Specifies that the animation should begin ⟨*time offset*⟩ many seconds after the ⟨*sync base id*⟩ with the given ⟨*type*⟩ has begun. Here, the ⟨*sync base id*⟩ must have been obtained using `\pgfsys@new@id`.

The idea behind a sync base is that you setup an animation and name it, other animations can start alongside this animation. An animation whose sole purpose is to orchestrate other animations in this way is called a *sync base*.

`\pgfsysanimkeysyncend{`⟨*sync base id*⟩`}{`⟨*type*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

`\pgfsys@animation@syncend{`⟨*sync base id*⟩`}{`⟨*type*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

Works like `\pgfsysanimkeysyncbegin` only the animation begin (or ends) when the sync base ends.

`\pgfsysanimkeyevent{`⟨*id*⟩`}{`⟨*type*⟩`}{`⟨*event name*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

`\pgfsys@animation@event{`⟨*id*⟩`}{`⟨*type*⟩`}{`⟨*event name*⟩`}{`⟨*time offset*⟩`}{`⟨*begin or end*⟩`}`

Specifies that the animation should begin (or end) ⟨*time offset*⟩ many seconds after a certain *event* has occurred. Which events are possible depends on the specific output language, here are the events currently supported in SVG:

- `click` occurs when the object with the given ⟨*id*⟩ and ⟨*type*⟩ has been clicked.
- `focusin` and `focusout` occur when the focus enters or leaves the object.
- `mouseup`, `mousedown`, `mouseover`, `mousemove`, and `mouseout` occur when the mouse is pressed up or down on the object, moved onto the object, moved over the object, or moved off the object.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyevent{\nodeid}{}{mouseup}{}{begin}
  \pgfsysanimate{translate} }
```




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyevent{\nodeid}{}{mousedown}{}{begin}
  \pgfsysanimate{translate} }
```




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyevent{\nodeid}{}{mouseover}{}{begin}
  \pgfsysanimate{translate} }
```




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyevent{\nodeid}{}{mousemove}{}{begin}
  \pgfsysanimate{translate} }
```

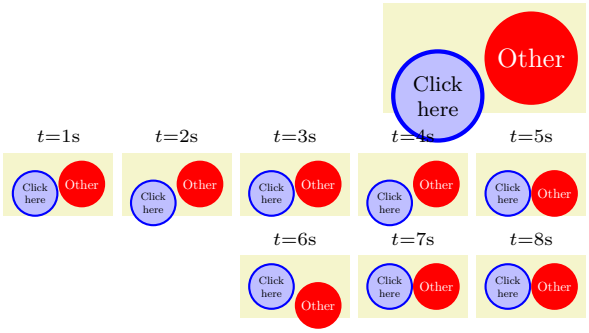



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyevent{\nodeid}{}{mouseout}{}{begin}
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyrepeatevent{⟨*id*⟩}{⟨*type*⟩}{⟨*repeat count*⟩}{⟨*time offset*⟩}{⟨*begin or end*⟩}

\pgfsys@animation@repeat@event{⟨*id*⟩}{⟨*type*⟩}{⟨*repeat count*⟩}{⟨*time offset*⟩}{⟨*begin or end*⟩}

The animation begins (or end) with a certain offset when another animation has reached a certain repeat count.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-5mm}
  \pgfsysanimkeyrepeatdur{5}
  \pgfsys@new@id{\animationid}
  \pgfsys@use@id{\animationid}
  \pgfsysanimate{translate}
  \global\let\animationid\animationid }
\tikz {
  \pgfidrefnextuse{\objid}{other}
  \pgfsysanimkeyrepeatevent{\animationid}{}{2}{0}{begin}
  \pgfsysanimkeysnapshotstart{4}
  \pgfsysanimkeywhom{\objid}{}
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-5mm}
  \pgfsysanimate{translate}
  \node [fill=red, text=white, circle] (other) {Other}; }
```

\pgfsysanimkeyaccesskey{⟨*character*⟩}{⟨*time offset*⟩}{⟨*begin or end*⟩}

\pgfsys@animation@accesskey{⟨*character*⟩}{⟨*time offset*⟩}{⟨*begin or end*⟩}

Begin or end the animation when a certain key is pressed. Note that this event may not be supported by some browsers for security reasons (prevent key loggers).




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyaccesskey{s}{}{begin}
  \pgfsysanimate{translate}
}
```

123.10 Commands for Specifying Timing: Restart Behaviour

\pgfsysanimkeyrestartalways

\pgfsys@animation@restart@always

Defines that the animation can be restarted at any time. This is the default.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrestartalways
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyrestartnever

\pgfsys@animation@restart@never

Defines that the animation cannot be restarted once it has run.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrestartnever
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyrestartwhennotactive

\pgfsys@animation@restart@whennotactive

Defines that the animation cannot be restarted while it is running.

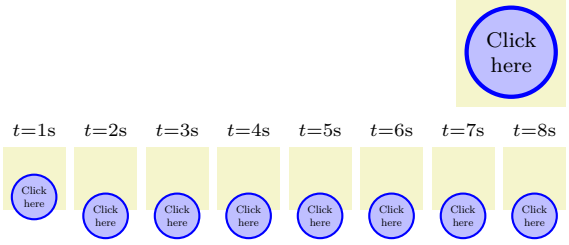



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyrestartwhennotactive
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyfreezeatend

\pgfsys@animation@freezeatend

When an animation ends, the question is whether the "effect" of the animation (like changing a color or translating the coordinate system) should disappear or "remain in force". Using this key, you specify that at the end of the animation the last value of the attributes stays in effect.

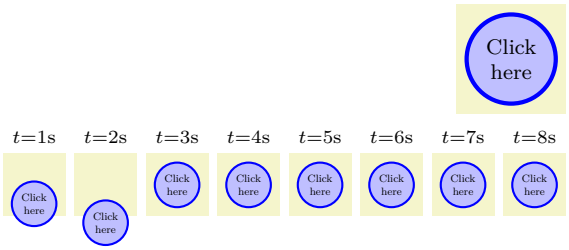



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyfreezeatend
  \pgfsysanimate{translate} }
```

\pgfsysanimkeyremoveatend

\pgfsys@animation@removeatend

The opposite of \pgfsysanimkeyfreezeatend. This is the default.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-1cm}
  \pgfsysanimkeyremoveatend
  \pgfsysanimate{translate} }
```

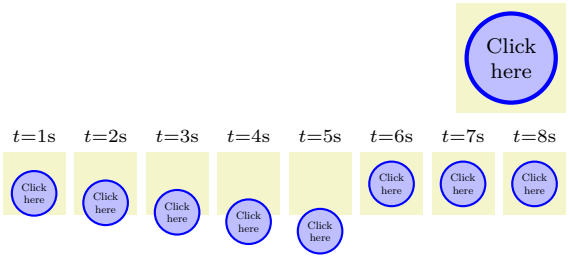
123.11 Commands for Specifying Accumulation

Animations specify how an attribute of an object changes over time. When more than one animation changes the same value at the same time, the last value given for the attribute "wins", except for animations of the canvas, which always accumulate. Additionally, when a repeat is specified for an attribute, during each repeat the values can add up:

\pgfsysanimkeyaccumulate

\pgfsys@animation@accumulate

Specifies that each repeat of an animation works as if the last values attained during previous repeats are added to the current value.

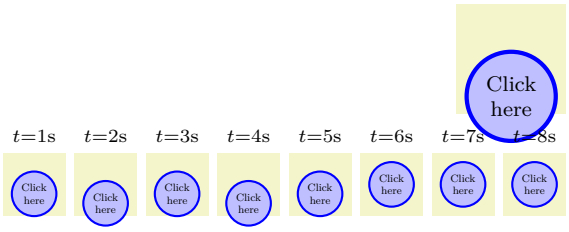



```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-5mm}
  \pgfsysanimkeyaccumulate
  \pgfsysanimkeyrepeatdur{5}
  \pgfsysanimate{translate}  }
```

\pgfsysanimkeynoaccumulate

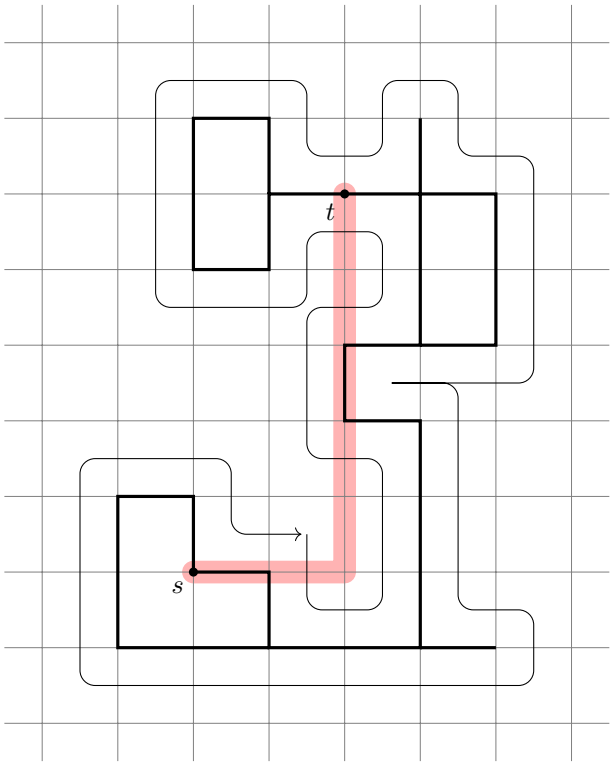
\pgfsys@animation@noaccumulate

Specifies that each repeat resets the to-be-animated value. This is the default.




```
\usetikzlibrary {animations}
\animationexample{node}{}{
  \pgfsysanimkeytime{0}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{0cm}
  \pgfsysanimkeytime{2}{1}{1}{0}{0}
  \pgfsysanimvaltranslate{0cm}{-5mm}
  \pgfsysanimkeynoaccumulate
  \pgfsysanimkeyrepeatdur{5}
  \pgfsysanimate{translate}  }
```

Part XI
References and Index




```
\begin{tikzpicture}
  \draw[line width=0.3cm,color=red!30,line cap=round,line join=round] (0,0)--(2,0)--(2,5);
  \draw[help lines] (-2.5,-2.5) grid (5.5,7.5);
  \draw[very thick] (1,-1)--(-1,-1)--(-1,1)--(0,1)--(0,0)--
    (1,0)--(1,-1)--(3,-1)--(3,2)--(2,2)--(2,3)--(3,3)--
    (3,5)--(1,5)--(1,4)--(0,4)--(0,6)--(1,6)--(1,5)
    (3,3)--(4,3)--(4,5)--(3,5)--(3,6)
    (3,-1)--(4,-1);
  \draw[below left] (0,0) node(s){$s$};
  \draw[below left] (2,5) node(t){$t$};
  \fill (0,0) circle [radius=0.06cm] (2,5) circle [radius=0.06cm];
  \draw[->,rounded corners=0.2cm,shorten >=2pt]
    (1.5,0.5)-- ++(0,-1)-- ++(1,0)-- ++(0,2)-- ++(-1,0)-- ++(0,2)-- ++(1,0)--
    ++(0,1)-- ++(-1,0)-- ++(0,-1)-- ++(-2,0)-- ++(0,3)-- ++(2,0)-- ++(0,-1)--
    ++(1,0)-- ++(0,1)-- ++(1,0)-- ++(0,-1)-- ++(1,0)-- ++(0,-3)-- ++(-2,0)--
    ++(1,0)-- ++(0,-3)-- ++(1,0)-- ++(0,-1)-- ++(-6,0)-- ++(0,3)-- ++(2,0)--
    ++(0,-1)-- ++(1,0);
\end{tikzpicture}
```

Index

This index only contains automatically generated entries. A good index should also contain carefully selected keywords. This index is not a good index.

' key, 246
( ) group math operators, 1034
* infix math operator, 1033
* plot mark, 757
|| math operator, 1034
| plot mark, 758
+ infix math operator, 1033
+ plot mark, 757
- infix math operator, 1033
- plot mark, 758
- prefix math operator, 1033
-- key, 294
-- path operation, 154
-| path operation, 155
|- path operation, 155
--plot path operation, 341
-> key, 294
-!- key, 294
.. path operation, 155
/ infix math operator, 1033
: path operation, 169
< infix math operator, 1033
<- key, 294
<-> key, 294
<= infix math operator, 1034
== infix math operator, 1033
> infix math operator, 1033
> key, 209
>= infix math operator, 1033
? : conditional math operators, 1033
__add (Lua), 514
__div (Lua), 514
__mul (Lua), 514
__sub (Lua), 514
__tostring (Lua), 503
__unm (Lua), 514
@dec sep mark key, 1062
@sci exponent mark key, 1062
" " group math operators, 1034
^ infix math operator, 1033
{ } array math operators, 1034
! postfix math operator, 1033
! prefix math operator, 1034
!= infix math operator, 1033
[ ] array access math operators, 1034
⟨*axis name*⟩ key, 874
⟨*axis system name*⟩ key, 919
⟨*chain name*⟩-begin node, 603
⟨*chain name*⟩-end node, 603
⟨*color class name*⟩ key, 292
⟨*legend name*⟩ key, 945
⟨*meaning of character*⟩ key, 980
⟨*shape name*⟩ option, 224
⟨*visualizer name*⟩ key, 923
&& infix math operator, 1034
10-pointed star plot mark, 758
1000 sep key, 1059
1000 sep in fractionals key, 1059
16 on 1 layout, 1018
2 on 1 layout, 1018
3d library, 564
3d view key, 738
4 on 1 layout, 1018
6 on 1 layout, 1018
8 on 1 layout, 1018

about key, 891
about strategy key, 891
above key, 237, 238
above delimiter key, 712
above left key, 238, 241
above right key, 238, 242
abs math function, 1037
absolute key, 249
ac source key, 629
accepting key, 573
accepting above key, 574
accepting below key, 574
accepting by arrow key, 573
accepting by double key, 573
accepting left key, 574
accepting right key, 574
accepting text key, 573
accepting where key, 574
acos math function, 1041
.activate family handler, 998
active families key, 999
active families and known key, 999
active families or descendants of key, 999
active families or no family key, 999
active families or no family DEBUG key, 999
.add handler, 990
add (Lua), 501, 537
add math function, 1035
.add code handler, 987
add exception key, 984
.add style handler, 988
addHandler (Lua), 523
addNecklaceCircleHint (Lua), 523
addOne (Lua), 537
addToVertexOptions (Lua), 553
adjustable key, 630
after creation key, 864
after survey key, 863
after visualization key, 863
\afterdecoration, 1118
alias key, 224
align key, 233
align here key, 436
all date test, 1010
all axes key, 874
allow inside edges key, 442
allow upside down key, 247
along key, 1223

also at key, 895
ampere key, 630
amperemeter key, 629
ampersand replacement key, 326
amplitude key, 637
\anchor, 1148
anchor (Lua), 506, 522
anchor key, 139, 237, 326, 652, 702
anchor at key, 428
anchor here key, 428
anchor node key, 427
\anchorborder, 1149
and key, 1000
and math function, 1042
and gate key, 619, 620
and gate IEC symbol key, 624
angle key, 137, 140, 194, 564, 637, 733–735
angle pic type, 568
angle eccentricity key, 568
angle radius key, 568
angle' key, 194
angles library, 568
animate key, 382
Animation attributes
  color, 395
  dash, 396
  dash phase, 396
  draw, 394
  fill, 394
  fill opacity, 395
  line width, 396
  opacity, 395
  path, 397
  position, 403
  rotate, 399
  scale, 399
  shift, 400
  stage, 395
  stroke opacity, 395
  text, 395
  view, 404
  visible, 395
  xscale, 399
  xshift, 399
  xskew, 399
  xslant, 399
  yscale, 399
  yshift, 399
  yskew, 399
  yslant, 399
Animation attributes (basic layer)
  dash, 1218
  draw, 1216
  draw opacity, 1217
  fill, 1215
  fill opacity, 1217
  line width, 1218
  motion, 1222
  opacity, 1217
  path, 1219
  rotate, 1221
  scale, 1221
  skew x, 1222
  skew y, 1222
  stage, 1218
  translate, 1222
  view, 1224
  visible, 1218
  xskew, 1221
  yskew, 1221
Animation attributes (system layer)
  dash, 1260
  fillcolor, 1258
  fillopacity, 1257
  linewidth, 1260
  motion, 1263
  opacity, 1257
  path, 1258
  rotate, 1263
  scale, 1263
  skewx, 1263
  skewy, 1263
  strokecolor, 1258
  strokeopacity, 1258
  translate, 1262
  viewbox, 1264
  visibility, 1258
animations library, 380
animations module, 1209
annotation key, 721
annotation arrow key, 615, 628
.append handler, 990
append after command key, 153
.append code handler, 988
append filtered to key, 998
.append style handler, 989
appendArc (Lua), 517
appendArcTo (Lua), 518
appendClosepath (Lua), 516
appendCurveto (Lua), 516
appendLineto (Lua), 516
appendMoveto (Lua), 516
apply (Lua), 513
approximate remote forces key, 477
Arc (Lua), 506
arc (Lua), 501
arc key, 195
arc path operation, 158
Arc Barb arrow tip, 212
Arc:
  eventIndex (Lua), 508
  headAnchorForArcPath (Lua), 509
  options (Lua), 508
  optionsAccumulated (Lua), 508
  optionsArray (Lua), 507
  pointCloud (Lua), 508
  setPolylinePath (Lua), 510
  spanPriority (Lua), 509
  sync (Lua), 509
  syntacticTailAndHead (Lua), 508
  tailAnchorForArcPath (Lua), 509
arg1 key, 864
arg1 from key key, 864
arg1 handle from key key, 864
array math function, 1045
\arrow, 649

arrow box shape, 813
arrow box arrows key, 814
arrow box east arrow key, 814
arrow box head extend key, 814
arrow box head indent key, 814
arrow box north arrow key, 814
arrow box shaft width key, 814
arrow box south arrow key, 814
arrow box tip angle key, 814
arrow box west arrow key, 814
Arrow tips
Arc Barb, 212
Bar, 212
Bracket, 212
Butt Cap, 219
Circle, 216
Classical TikZ Rightarrow, 215
Computer Modern Rightarrow, 215
Diamond, 216
direction ee, 630
Ellipse, 216
Fast Round, 219
Fast Triangle, 220
Hooks, 212
Implies, 215
Kite, 217
LaTeX, 217
Latex, 217
Parenthesis, 213
Rays, 221
Rectangle, 218
Round Cap, 220
Square, 218
Stealth, 218
Straight Barb, 213
Tee Barb, 214
To, 216
Triangle, 219
Triangle Cap, 220
Turned Square, 219
\arrowreversed, 649
arrows key, 189, 209, 398, 1219
arrows option, 189
arrows.meta library, 210
as key, 280
asin math function, 1041
aspect key, 637, 787, 800
assume math mode key, 1062
asterisk plot mark, 758
at key, 157, 223, 895, 1083
at end key, 247
at end survey key, 863
at end visualization key, 863
at least date test, 1011
at most date test, 1011
at start key, 247
at start survey key, 863
at start visualization key, 863
atan math function, 1041
atan2 math function, 1041
\attribute, 1067
attribute key, 385, 668
auto key, 245
auto corner on length key, 1112
auto end on length key, 1112
automata library, 571
aux in dpth key, 671
axiom key, 701
axis shading, 776
axis layer key, 910
axis option/anchor at max key, 917
axis option/anchor at min key, 917
axis options/attribute key, 874
axis options/degrees key, 972
axis options/exponential steps key, 905
axis options/function key, 877
axis options/goto key, 908
axis options/goto pos key, 909
axis options/grid key, 889
axis options/include value key, 875
axis options/label key, 880
axis options/length key, 879
axis options/linear steps key, 905
axis options/logarithmic key, 878
axis options/max value key, 876
axis options/min value key, 875
axis options/padding key, 911
axis options/padding max key, 911
axis options/padding min key, 910
axis options/power unit length key, 880
axis options/radians key, 972
axis options/scaling key, 876
axis options/scaling/default key, 878
axis options/tick placement strategy key, 906
axis options/ticks key, 888
axis options/ticks and grid key, 889
axis options/unit length key, 879
axis options/unit vector key, 881
axis options/unit vectors key, 972
axis options/visualize axis key, 908
axis options/visualize grid key, 911
axis options/visualize label key, 916
axis options/visualize ticks key, 913

babel library, 576
back key, 848
background code key, 264
background grid key, 579
background rectangle key, 578
background top key, 580
\backgroundpath, 1150
backgrounds library, 577
backward diode key, 629
badness warnings for centered text key, 235
balanced minimum evolution key, 484
balanced nearest neighbour interchange key, 485
ball plot mark, 347
ball shading, 776
ball color key, 777
Bar arrow tip, 212
bar interval shift key, 754
bar interval width key, 754
bar shift key, 753
bar width key, 753
barycentric coordinate system, 138
base key, 404, 1083, 1213
base left key, 242

base right key, 242
baseline key, 129
battery key, 629
battery IEC shape, 635
before background key, 632, 633
before creation key, 863
before survey key, 863
before visualization key, 863
\beforebackgroundpath, 1150
\beforedecoration, 1118
\beforeforegroundpath, 1150
begin key, 1225
begin on key, 1225
begin snapshot key, 1215
beginGraphDrawingScope (Lua), 552
\beginpgfgraphicnamed, 1179
behind path key, 223
\behindbackgroundpath, 1150
\behindforegroundpath, 1150
.belongs to family handler, 998
below key, 238, 240
below delimiter key, 712
below left key, 238, 242
below right key, 238, 242
bend key, 161, 204
bend angle key, 841
bend at end key, 162
bend at start key, 162
bend left key, 840
bend pos key, 161
bend right key, 841
bending library, 202
bent decoration, 640
between date test, 1011
bilinear interpolation shading, 777
bin math function, 1044
binary tree layout key, 461
bind (Lua), 552
Binding (Lua), 557
Binding:
    createVertex (Lua), 559
    declareCallback (Lua), 558
    everyEdgeCreation (Lua), 559
    everyVertexCreation (Lua), 558
    renderCollection (Lua), 558
    renderCollectionStartKind (Lua), 558
    renderCollectionStopKind (Lua), 558
    renderEdge (Lua), 559
    renderEdgesStart (Lua), 559
    renderEdgesStop (Lua), 559
    renderStart (Lua), 558
    renderStop (Lua), 558
    renderVertex (Lua), 558
    renderVerticesStart (Lua), 558
    renderVerticesStop (Lua), 558
BindingToPGF (Lua), 560
bk key, 849
blend group key, 356
blend mode key, 355
boolean expected key, 995
border decoration, 641
bottom key, 1083
bottom color key, 776
bottom left key, 732, 736
bounding box key, 736
boundingBox (Lua), 505, 515, 517
brace decoration, 642
Bracket arrow tip, 212
branch down key, 302
branch down sep key, 305
branch left key, 302
branch left sep key, 304
branch right key, 302
branch right sep key, 304
branch up key, 301
branch up sep key, 305
breadth first spanning tree key, 465
break contact key, 629
break contact IEC shape, 636
breakdown diode key, 629
breakdown diode IEC shape, 633
\breakforeach, 1008
bricks pattern, 730
buffer gate key, 620
buffer gate IEC symbol key, 624
bulb key, 629
bumps decoration, 640
Butt Cap arrow tip, 219
butterfly key, 316
butterfly/from key, 317
butterfly/level key, 317
butterfly/to key, 317
by key, 144

\c@pgf@counta, 1228
\c@pgf@countb, 1228
\c@pgf@countc, 1228
\c@pgf@countd, 1228
calc library, 147, 581
\calendar, 582
calendar library, 582
callout absolute pointer key, 824
callout pointer arc key, 825
callout pointer end size key, 827
callout pointer segments key, 827
callout pointer shorten key, 824
callout pointer start size key, 826
callout pointer width key, 825
callout relative pointer key, 824
Cantor set decoration, 666
canvas coordinate system, 135
canvas is plane key, 565
canvas is xy plane at z key, 565
canvas is xz plane at y key, 565
canvas is yx plane at z key, 565
canvas is yz plane at x key, 565
canvas is zx plane at y key, 565
canvas is zy plane at x key, 566
canvas polar coordinate system, 137
cap angle key, 221
capacitor key, 629
capacitor IEC shape, 635
Cartesian placement key, 300
.cd handler, 986
ceil math function, 1038
cells key, 322
centered key, 238

chain default direction key, 600
chain polar shift key, 305
chain shift key, 300
\chainin, 603
chains library, 600
chamfered rectangle shape, 830
chamfered rectangle angle key, 830
chamfered rectangle corners key, 831
chamfered rectangle sep key, 831
chamfered rectangle xsep key, 830
chamfered rectangle ysep key, 831
checkerboard pattern, 730
checkerboard light gray pattern, 731
child path operation, 331
child anchor key, 340
children (Lua), 512
children are tokens key, 745
childrenOfKind (Lua), 512
Circle arrow tip, 216
circle path operation, 157
circle shape, 785
circle connection bar decoration, 718
circle connection bar key, 719
circle connection bar switch color key, 720
circle ee shape, 631
circle solidus shape, 817
circle split shape, 816
circle through key, 844
circle with fuzzy edge 10 percent fading, 683
circle with fuzzy edge 15 percent fading, 683
circle with fuzzy edge 20 percent fading, 683
circuit declare annotation key, 615
circuit declare symbol key, 610
circuit declare unit key, 614
circuit ee key, 626
circuit ee IEC key, 626
circuit handle symbol key, 611
circuit logic key, 618
circuit logic CDH key, 619
circuit logic IEC key, 618
circuit logic US key, 618
circuit symbol filled key, 616
circuit symbol lines key, 616
circuit symbol open key, 616
circuit symbol size key, 610
circuit symbol unit key, 610
circuit symbol wires key, 616
circuits key, 610
circuits library, 609
circuits.ee library, 626
circuits.ee.IEC library, 626
circuits.logic library, 617
circuits.logic.CDH library, 619
circuits.logic.IEC library, 618
circuits.logic.US library, 618
circular graph drawing library, 481
circular drop shadow key, 782
circular glow key, 783
circular placement key, 305
circular sector shape, 798
circular sector angle key, 799
class (Lua), 536
class key, 122, 863
Class object
    copy, 1070
Class object, 1070
Class signal
    connect, 1071
    emit, 1071
    signal, 1071
Class signal, 1070
Classes
    object, 1070
    signal, 1070
Classical TikZ Rightarrow arrow tip, 215
clear (Lua), 516
clear < key, 290
clear > key, 290
\clip, 171
clip key, 185
clique key, 314
clockwise key, 307
clockwise from key, 846
clone (Lua), 513, 516
closepath code key, 644
cloud shape, 802
cloud callout shape, 826
cloud ignores aspect key, 802
cloud puff arc key, 802
cloud puffs key, 802
cm key, 378
coarsen key, 478
.code handler, 986
code key, 263, 732
.code 2 args handler, 987
.code args handler, 987
.code n args handler, 987
coil decoration, 640
collapse (Lua), 503
Collection (Lua), 511
Collection.
    new (Lua), 512
Collection:
    children (Lua), 512
    childrenOfKind (Lua), 512
    descendants (Lua), 512
    descendantsOfKind (Lua), 513
color animation attribute, 395
color key, 172, 197
color class key, 292
color option option, 172
color wheel shading, 777
color wheel black center shading, 778
color wheel white center shading, 778
\colorcurrentmixin, 1023
colored tokens key, 746
colormixin environment, 1023
column ⟨*number*⟩ key, 323
column sep key, 320
common key, 894
complete bipartite key, 315
component align key, 436
component direction key, 435
component distance key, 425
component order key, 434
component packing key, 441

component sep key, 425
components go down key, 440
components go down absolute left aligned key, 440
components go down absolute right aligned key, 440
components go down center aligned key, 440
components go down left aligned key, 440
components go down right aligned key, 440
components go left key, 440
components go left absolute bottom aligned key, 440
components go left absolute top aligned key, 439
components go left bottom aligned key, 439
components go left center aligned key, 440
components go left top aligned key, 439
components go right key, 439
components go right absolute bottom aligned key, 439
components go right absolute top aligned key, 439
components go right bottom aligned key, 439
components go right center aligned key, 439
components go right top aligned key, 439
components go up key, 440
components go up absolute left aligned key, 440
components go up absolute right aligned key, 440
components go up center aligned key, 440
components go up left aligned key, 440
components go up right aligned key, 440
componentwise key, 434
compute step key, 906
Computer Modern Rightarrow arrow tip, 215
concat (Lua), 518
concept key, 714
concept color key, 715, 716
concept connection key, 717
connect (Lua), 502
connect method, 1071
connect spies key, 836
const plot key, 349
const plot mark left key, 349
const plot mark mid key, 350
const plot mark right key, 349
contact key, 629
contains (Lua), 501
continue branch key, 605
continue chain key, 601
controls key, 842
convergence tolerance key, 476
convert (Lua), 560
cooling factor key, 475
Coordinate (Lua), 513
\coordinate, 226
coordinate path operation, 226
coordinate tikz math function, 705
Coordinate systems
    barycentric, 138
    canvas, 135
    canvas polar, 137
    node, 139
    perpendicular, 143
    tangent, 141
    three point perspective, 738
    tpp, 738
    visualization, 862
    xy polar, 138
    xyz, 136
    xyz cylindrical, 564
    xyz polar, 137
    xyz spherical, 564
Coordinate.
    __add (Lua), 514
    __div (Lua), 514
    __mul (Lua), 514
    __sub (Lua), 514
    __unm (Lua), 514
    boundingBox (Lua), 515
    new (Lua), 513
Coordinate:
    apply (Lua), 513
    clone (Lua), 513
    moveTowards (Lua), 514
    norm (Lua), 514
    normalize (Lua), 514
    normalized (Lua), 515
    scale (Lua), 514
    shift (Lua), 513
    shiftByCoordinate (Lua), 514
    unshift (Lua), 514
    unshiftByCoordinate (Lua), 514
coordinates (Lua), 517
copy (Lua), 535
copy method, 1070
copy shadow key, 781
correct forbidden sign shape, 801
cos math function, 1040
cos path operation, 162
cosec math function, 1040
cosh math function, 1045
cot math function, 1041
coulomb key, 630
count key, 1007
counterclockwise key, 307
counterclockwise from key, 846
createEdge (Lua), 530, 554
createEvent (Lua), 554
createVertex (Lua), 529, 553, 559
cross marks style sheet, 938
cross out shape, 828
crosses decoration, 651
crosshatch pattern, 730
crosshatch dots pattern, 730
crosshatch dots gray pattern, 731
crosshatch dots light steel blue pattern, 731
css key, 122
cube folding pic type, 725
cube truncated folding pic type, 725
cuboctahedron folding pic type, 727
cuboctahedron truncated folding pic type, 727
current bounding box node, 1144
current direction key, 629
current direction' key, 629
current page node, 1145
current path bounding box node, 1144
current point is local key, 147

current source key, 629
current subpath start node, 1144
curve to key, 839
curveto decoration, 641
curveto code key, 644
curvilinear library, 1169
cut policy key, 442
cutAtBeginning (Lua), 517
cutAtEnd (Lua), 517
cutEdges (Lua), 522
cycle key, 314
cylinder shape, 800
cylinder body fill key, 800
cylinder end fill key, 800
cylinder uses custom fill key, 800

dart shape, 797
dart tail angle key, 797
dart tip angle key, 797
dash animation attribute, 396
dash basic layer animation attribute, 1218
dash key, 175
dash system layer animation attribute, 1260
dash dot key, 176
dash dot dot key, 176
dash expand off key, 175
dash pattern key, 174
dash phase animation attribute, 396
dash phase key, 175
dashed key, 175
data (data visualization), 859
data bounding box node, 862
Data formats, *see* Formats
data group (data visualization), 861
/data point/
    outlier, 926
    set, 922
data point (data visualization), 860
data point key, 860
Data visualization
    data, 859
    data group, 861
    data point, 860
    info, 862
    info', 862
    scope, 861
data visualization bounding box node, 862
\datavisualization, 858
datavisualization library, 855
datavisualization.formats.functions library, 867
datavisualization.polar library, 964
Date tests
    all, 1010
    at least, 1011
    at most, 1011
    between, 1011
    day of month, 1011
    Easter, 1011
    end of month, 1011
    equals, 1010
    Friday, 1010
    Monday, 1010
    Saturday, 1010
    Sunday, 1010
    Thursday, 1010
    Tuesday, 1010
    Wednesday, 1010
    weekend, 1010
    workday, 1010
dates key, 582
day code key, 584
day list downward key, 590
day list left key, 591
day list right key, 591
day list upward key, 591
day of month date test, 1011
day text key, 585
day xshift key, 583
day yshift key, 583
dc source key, 629
.deactivate family handler, 998
debug (Lua), 534
dec sep key, 1058
decimal about strategy key, 893
declare (Lua), 524
declare key, 298
declare function key, 1051
declare_algorithm (Lua), 527
declare_collection_kind (Lua), 528
declare_parameter (Lua), 525
declareCallback (Lua), 558
decompose (Lua), 522
decorate key, 369
decorate path operation, 367
\decoration, 1118
decoration key, 367
Decorations
    bent, 640
    border, 641
    brace, 642
    bumps, 640
    Cantor set, 666
    circle connection bar, 718
    coil, 640
    crosses, 651
    curveto, 641
    expanding waves, 642
    footprints, 649
    Koch curve type 1, 665
    Koch curve type 2, 665
    Koch snowflake, 665
    lineto, 638
    markings, 645
    moveto, 642
    random steps, 639
    saw, 639
    shape backgrounds, 651
    show path construction, 643
    snake, 641
    straight zigzag, 638
    text along path, 655
    text effects along path, 657
    ticks, 642
    triangles, 651
    waves, 643
    zigzag, 639
decorations library, 367

decorations module, 1108
decorations.footprints library, 649
decorations.fractals library, 664
decorations.markings library, 644
decorations.pathmorphing library, 638
decorations.pathreplacing library, 641
decorations.shapes library, 651
decorations.text library, 654
.default handler, 986
default edge kind key, 294
default edge operator key, 297
defaults key, 732
\deferredanchor, 1149
defined key, 1000
deg math function, 1040
delta angle key, 159
densely dash dot key, 176
densely dash dot dot key, 176
densely dashed key, 176
densely dotted key, 175
depth math function, 1046
depth first cycle removal key, 470
depth first spanning tree key, 466
dequeue (Lua), 538
descendants (Lua), 512
descendantsOfKind (Lua), 513
desired at key, 425
desired child index key, 464
Diamond arrow tip, 216
diamond plot mark, 758
diamond shape, 786
diamond split shape, 819
diamond* plot mark, 758
Digraph (Lua), 498
Digraph.
    new (Lua), 501
Digraph:
    __tostring (Lua), 503
    add (Lua), 501
    arc (Lua), 501
    collapse (Lua), 503
    connect (Lua), 502
    contains (Lua), 501
    disconnect (Lua), 502
    expand (Lua), 503
    incoming (Lua), 502
    orderIncoming (Lua), 502
    orderOutgoing (Lua), 502
    outgoing (Lua), 502
    reconnect (Lua), 502
    remove (Lua), 501
    sortIncoming (Lua), 502
    sortOutgoing (Lua), 502
    sync (Lua), 503
dim math function, 1045
diode key, 629
direction axis key, 912
direction ee arrow tip, 630
direction ee shape, 631
direction ee arrow key, 631
direction info key, 630
disable dependency files key, 674
disconnect (Lua), 502
distance key, 733–735, 842, 848
div math function, 1036
divide math function, 1036
doc (Lua), 494
doc.
    documentation (Lua), 494
    example (Lua), 495
    key (Lua), 494
    summary (Lua), 494
documentation (Lua), 494
dodecahedron folding pic type, 726
dodecahedron golden folding pic type, 727
dodecahedron rhombic folding pic type, 727
dodecahedron' folding pic type, 726
domain key, 343
Dots pattern, 735
dots pattern, 730
dotted key, 175
double key, 177
double arrow shape, 812
double arrow head extend key, 812
double arrow head indent key, 812
double arrow tip angle key, 812
double copy shadow key, 782
double distance key, 177
double distance between line centers key, 177
double equal sign distance key, 177
downsize ratio key, 479
draft package option, 1076, 1175
\draw, 171
draw animation attribute, 394
draw basic layer animation attribute, 1216
draw key, 172
draw opacity basic layer animation attribute, 1217
draw opacity key, 353
drop shadow key, 781

e math function, 1036
east fading, 683
Easter date test, 1011
.ecode handler, 987
.ecode 2 args handler, 987
.ecode args handler, 987
.ecode n args handler, 987
Edge (Lua), 510
edge key, 273
edge path operation, 256
edge from parent key, 339
edge from parent path operation, 338
edge from parent fork down key, 847
edge from parent fork left key, 847
edge from parent fork right key, 847
edge from parent fork up key, 847
edge from parent macro key, 340
edge from parent path key, 339
edge label key, 164, 274
edge label' key, 164, 274
edge node key, 164, 273
edge quotes key, 287
edge quotes center key, 287
edge quotes mid key, 287
Edge.
    new (Lua), 511
Edge:

headAnchorForEdgePath (Lua), 511
setPolylinePath (Lua), 511
tailAnchorForEdgePath (Lua), 511
edges key, 273
edges behind nodes key, 448
electric charge key, 477
electric force order key, 477
Ellipse arrow tip, 216
ellipse path operation, 158
ellipse shape, 787
ellipse callout shape, 825
ellipse split shape, 818
emit method, 1071
⟨*empty*⟩ path operation, 154
empty nodes key, 281
end key, 1225
end angle key, 159
end of month date test, 1011
end on key, 1226
end radius key, 637
endGraphDrawingScope (Lua), 553
\endpgfgraphicnamed, 1180
enqueue (Lua), 538
entity key, 667
entry key, 383, 1212
entry control key, 1212
Environments
colormixin, 1023
pgfdecoration, 1114, 1117
pgfidscope, 1084
pgfinterruptboundingbox, 1082
pgfinterruptpath, 1081
pgfinterruptpicture, 1081
pgfkeysinterruptkeyfilter, 1001
pgflowlevelscope, 1165
pgfmetadecoration, 1120
pgfonlayer, 1193, 1194
pgfpicture, 1077, 1079
pgfscope, 1080
pgftransparencygroup, 1207, 1208
pgfviewboxscope, 1165, 1166
scope, 131, 132
tikzfadingfrompicture, 358, 359
tikzpicture, 128, 130
equal math function, 1042
equals date test, 1010
equals key, 1000
er library, 667
/errors/
boolean expected, 995
unknown choice value, 995
unknown key, 995
value forbidden, 995
value required, 995
escape key, 121
.estore in handler, 990
.estyle handler, 988
.estyle 2 args handler, 988
.estyle args handler, 989
euro about strategy key, 892
evaluate key, 703, 1006
.evaluated handler, 991
even odd rule key, 180
Event (Lua), 520
Event.
new (Lua), 520
eventIndex (Lua), 508
events/begin key, 407
events/click key, 406
events/delay key, 408
events/end key, 407
events/event key, 406
events/focus in key, 408
events/focus out key, 408
events/key key, 408
events/mouse down key, 406
events/mouse move key, 407
events/mouse out key, 407
events/mouse over key, 407
events/mouse up key, 407
events/of key, 406
events/of next key, 406
events/repeat key, 408
every ⟨*axis system name*⟩ key, 919
every ⟨*part name*⟩ node part key, 231
every ⟨*shape*⟩ node key, 225
every above delimiter key, 712
every accepting by arrow key, 574
every annotation key, 722
every attribute key, 668
every axis key, 910
every below delimiter key, 712
every calendar key, 582
every cell key, 322
every child key, 334
every child node key, 334
every circle key, 158
every circle connection bar key, 719
every circuit ee key, 626
every circuit logic key, 618
every circuit symbol key, 611
every concept key, 715
every cut key, 724
every data key, 860, 870
every data set label key, 943
every day key, 585
every decoration key, 1116
every delimiter key, 712
every edge key, 257
every edge quotes key, 257
every entity key, 667
every even column key, 323
every even row key, 323
every extra concept key, 715
every fit key, 684
every fold key, 724
every graph key, 273
every grid key, 897
every info key, 613
every initial by arrow key, 573
every join key, 604
every label key, 250
every label in data key, 943
every label in legend key, 956
every label quotes key, 254
every left delimiter key, 712

every loop key, 843
every major grid key, 898
every major ticks key, 899
every mark key, 347
every matrix key, 318
every mindmap key, 713
every minor grid key, 898
every minor ticks key, 899
every month key, 586
every new -- key, 277
every new -> key, 276
every new <- key, 277
every new <-> key, 277
every node key, 225
every odd column key, 323
every odd row key, 323
every on background layer key, 577
every on chain key, 603
every outer matrix key, 318
every path key, 153
every pic key, 264
every pic quotes key, 266
every picture key, 130
every pin key, 251
every pin edge key, 251
every pin quotes key, 254
every place key, 744
every plot key, 346
every relationship key, 668
every relative key, 1055
every right delimiter key, 712
every scientific axes key, 883
every scope key, 131
every shadow key, 781
every spy in node key, 834
every spy on node key, 835
every state key, 572
every subgraph node key, 445
every subminor grid key, 898
every subminor ticks key, 899
every ticks key, 899
every to key, 165, 166
every token key, 745
every transition key, 744
every visualizer key, 925
every year key, 587
everyEdgeCreation (Lua), 559
everyVertexCreation (Lua), 558
evolutionary unit length key, 487
example (Lua), 495
examples graph drawing library, 530
exec key, 995
execute after day scope key, 589
execute at begin cell key, 325
execute at begin day scope key, 588
execute at begin function key, 1052
execute at begin node key, 226
execute at begin picture key, 129
execute at begin scope key, 131
execute at begin to key, 166
execute at empty cell key, 325
execute at end cell key, 325
execute at end day scope key, 589
execute at end function key, 1052
execute at end node key, 226
execute at end picture key, 129
execute at end scope key, 131
execute at end to key, 166
execute before day scope key, 588
exit control key, 1212
exp math function, 1036
expand (Lua), 503
expand list key, 1007
.expand once handler, 991
.expand twice handler, 991
.expanded handler, 991
expanding waves decoration, 642
export key, 674
export next key, 674
extended binary tree layout key, 462
external library, 669
External Graphics
    Bounding Box Issues, 678, 1182
external info key, 681
extra concept key, 715

face 1 key, 724
face 2 key, 724
face 3 key, 724
face 4 key, 724
factorial math function, 1036
fading angle key, 361
fading transform key, 360
Fadings
    circle with fuzzy edge 10 percent, 683
    circle with fuzzy edge 15 percent, 683
    circle with fuzzy edge 20 percent, 683
    east, 683
    fuzzy ring 15 percent, 683
    north, 683
    south, 683
    west, 683
fadings library, 683
false key, 1000
false math function, 1043
farad key, 630
Fast Round arrow tip, 219
Fast Triangle arrow tip, 220
fd key, 848
few key, 893
figure list key, 676
figure name key, 673
File, *see* Packages and files
\fill, 171
fill animation attribute, 394
fill basic layer animation attribute, 1215
fill key, 178, 198
fill opacity animation attribute, 395
fill opacity basic layer animation attribute, 1217
fill opacity key, 354
fillcolor system layer animation attribute, 1258
\filldraw, 171
fillopacity system layer animation attribute, 1257
filtered pgfkeys library, 995
find (Lua), 534
find_min (Lua), 535
findVertexByName (Lua), 529

first key, 465
first char syntax key, 980
fit key, 684
fit library, 684
fit fading key, 360
fivepointed stars pattern, 730
fixed key, 1053
fixed point arithmetic key, 687
fixed relative key, 1056
fixed zerofill key, 1053
fixedpointarithmetic library, 687
flex key, 202
flex' key, 203
floor math function, 1038
folding library, 723
folding line length key, 724
font key, 232
foot angle key, 650
foot length key, 650
foot of key, 650
foot sep key, 650
footprints decoration, 649
for tikz math function, 706
forbidden sign shape, 802
force graph drawing library, 473
force remake key, 674
\foreach, 1003
foreach path operation, 166
foreground code key, 264
\foregroundpath, 1150
fork key, 387
format key, 859, 869
Formats
    function, 867
    named, 866
    table, 865
    TeX code, 867
forward key, 848
.forward to handler, 992
fourth key, 465
fpu key, 689
fpu library, 689
frac key, 1057
frac math function, 1039
frac denom key, 1057
frac shift key, 1058
frac TeX key, 1057
frac whole key, 1057
framed key, 578
freeze at end key, 1225
fresh nodes key, 278
Friday date test, 1010
function format, 867
function tikz math function, 707
fuzzy ring 15 percent fading, 683

gap around stream point key, 755
gcd math function, 1039
general shadow key, 780
generic circle IEC shape, 632
generic diode IEC shape, 632
.get handler, 989
get handle method, 1070
get id method, 1069
get new resource curie key, 766
get scope curie key, 767
get_current_options_table (Lua), 556
getDeclaredKeys (Lua), 554
getNecklaceHints (Lua), 523
gnuplot call key, 1190
\graph, 272
graph path operation, 272
/graph drawing/
    phylogenetic edge, 483
    phylogenetic inner node, 483
Graph drawing libraries
    circular, 481
    examples, 530
    force, 473
    layered, 468
    phylogenetics, 483
    routing, 488
    trees, 458
graphdrawing library, 420, 450
Graphic options and styles
    ⟨shape name⟩, 224
    arrows, 189
    color option, 172
Graphs
    subgraph C_n, 313
    subgraph Grid_n, 313
    subgraph I_n, 311
    subgraph I_nm, 312
    subgraph K_n, 312
    subgraph K_nm, 313
    subgraph P_n, 313
graphs library, 267
graphs.standard library, 311
gray scale style sheet, 940
greater math function, 1042
greedy cycle removal key, 471
grid path operation, 159
grid pattern, 730
grid layer key, 897
grid placement key, 302
gridded key, 579
ground key, 629
ground IEC shape, 636
group polar shift key, 305
group shift key, 300
grow key, 336, 430
grow cyclic key, 846
grow down key, 301
grow down sep key, 304
grow left key, 301
grow left sep key, 303
grow right key, 301
grow right sep key, 303
grow up key, 301
grow up sep key, 303
grow via three points key, 845
grow' key, 337, 431
growth function key, 338
growth parent anchor key, 338

half about strategy key, 892
halfcircle plot mark, 758
halfcircle* plot mark, 758

halfdiamond* plot mark, 758
halfsquare left* plot mark, 758
halfsquare right* plot mark, 758
halfsquare* plot mark, 758
handle active characters in code key, 576
handle active characters in nodes key, 576
handle new data sets in plots key, 1187
handle outlier points in plots key, 1187
handle undefined points in plots key, 1187
/handler config/
    only existing/
      add exception, 984
handler config, 983
handler config key, 983
/handlers/
    first char syntax/
      ⟨*meaning of character*⟩, 980
    first char syntax, 980
Handlers for keys, *see* Key handlers
handlers/empty number key, 692
handlers/invalid number key, 692
handlers/wrong lowlevel format key, 692
harpoon key, 196
Hatch pattern, 734
head anchor key, 442
head cut key, 442
headAnchorForArcPath (Lua), 509
headAnchorForEdgePath (Lua), 511
headline key, 866
heart plot mark, 758
height key, 833
height math function, 1046
help lines key, 161
henry key, 630
hertz key, 630
Hex math function, 1044
hex math function, 1044
high key, 910, 914
Hints (Lua), 523
Hints.
    addNecklaceCircleHint (Lua), 523
    getNecklaceHints (Lua), 523
home key, 848
Hooks arrow tip, 212
horizontal key, 430
horizontal line through key, 143
horizontal lines pattern, 730
horizontal lines dark blue pattern, 731
horizontal lines dark gray pattern, 731
horizontal lines gray pattern, 731
horizontal lines light blue pattern, 731
horizontal lines light gray pattern, 731
horizontal' key, 430
how key, 848
huge circuit symbols key, 610
huge mindmap key, 714
hyper key, 443

icopy (Lua), 535
icosahedron folding pic type, 728
icosidodecahedron folding pic type, 729
id (Lua), 536
id key, 122, 346, 385
if key, 587
if tikz math function, 707
if input segment is closepath key, 1112
\ifdate, 1013
\ifpgfallowupsidedownattime, 1161
\ifpgfgdgraphdrawingscopeactive, 452
\ifpgfmathmathunitsdeclared, 1033
\ifpgfmathunitsdeclared, 1032
\ifpgfrememberpicturepositiononpage, 1079
\ifpgfresetnontranslationsattime, 1161
\ifpgfshadingmodelcmyk, 1200
\ifpgfshadingmodelgray, 1201
\ifpgfshadingmodelrgb, 1200
\ifpgfslopedattime, 1161
\ifpgfsys@eorule, 1240
\ifpgfsys@transparency@group@isolated, 1244
\ifpgfsys@transparency@group@knockout, 1244
ifthenelse math function, 1043
ignore key, 998
ignore spaces key, 1052
imap (Lua), 535
Implies arrow tip, 215
in key, 839
in control key, 842
in distance key, 842
in front of path key, 223
in looseness key, 841
in max distance key, 842
in min distance key, 842
include external key, 678, 1182
incoming (Lua), 502
induced complete bipartite key, 316
induced cycle key, 314
induced independent set key, 314
induced path key, 315
inductor key, 629
inductor IEC shape, 634
infer tile bounding box key, 736
info (data visualization), 862
info key, 613
info sloped key, 614
info' (data visualization), 862
info' key, 613
info' sloped key, 614
\inheritanchor, 1151
\inheritanchorborder, 1151
\inheritbackgroundpath, 1151
\inheritbeforebackgroundpath, 1151
\inheritbeforeforegroundpath, 1151
\inheritbehindbackgroundpath, 1151
\inheritbehindforegroundpath, 1151
\inheritforegroundpath, 1151
\inheritsavedanchors, 1150
.initial handler, 989
initial key, 573
initial above key, 573
initial below key, 573
initial by arrow key, 573
initial by diamond key, 573
initial distance key, 573, 574
initial left key, 573
initial right key, 573
initial step length key, 475
initial text key, 573

initial where key, 573
inline key, 869
inner color key, 778
inner frame sep key, 578
inner frame xsep key, 578
inner frame ysep key, 578
inner sep key, 227, 1140
inner style order key, 324
inner style/cell key, 324
inner style/column key, 324
inner style/even odd column key, 324
inner style/even odd row key, 324
inner style/every cell key, 324
inner style/row key, 324
inner xsep key, 227, 1140
inner ysep key, 227, 1140
inputs key, 618, 619
insert path key, 153
inset key, 194
inset' key, 194
.install key filter handler, 997
.install key filter handler handler, 997
install only key, 690
int math function, 1038
int tikz math function, 705
int about strategy key, 893
int detect key, 1056
int trunc key, 1057
integer tikz math function, 705
InterfaceCore (Lua), 559
InterfaceCore.
    convert (Lua), 560
    topScope (Lua), 560
InterfaceToAlgorithms (Lua), 523
InterfaceToAlgorithms.
    addHandler (Lua), 523
    createEdge (Lua), 530
    createVertex (Lua), 529
    declare (Lua), 524
    findVertexByName (Lua), 529
InterfaceToDisplay (Lua), 551
InterfaceToDisplay.
    addToVertexOptions (Lua), 553
    beginGraphDrawingScope (Lua), 552
    bind (Lua), 552
    createEdge (Lua), 554
    createEvent (Lua), 554
    createVertex (Lua), 553
    endGraphDrawingScope (Lua), 553
    getDeclaredKeys (Lua), 554
    pushLayout (Lua), 554
    pushOption (Lua), 554
    pushSubgraphVertex (Lua), 553
    renderGraph (Lua), 555
    resumeGraphDrawingCoroutine (Lua), 552
    runGraphDrawingAlgorithm (Lua), 552
Internals
    \c@pgf@counta, 1228
    \c@pgf@countb, 1228
    \c@pgf@countc, 1228
    \c@pgf@countd, 1228
    \pgf@x, 1228
    \pgf@xa, 1228
    \pgf@xb, 1228
    \pgf@xc, 1228
    \pgf@y, 1228
    \pgf@ya, 1228
    \pgf@yb, 1228
    \pgf@yc, 1228
    \pgfutil@tempboxa, 1229
    \pgfutil@tempdima, 1228
    \pgfutil@tempdimb, 1228
    \r@pgf@reada, 1229
    \w@pgf@writea, 1228
intersections library, 143, 1094
intersectionsWith (Lua), 517
invert (Lua), 519
.is choice handler, 991
is descendant of key, 1000
.is family handler, 986, 998
.is if handler, 990
isEmpty (Lua), 538
iseven math function, 1039
isodd math function, 1039
isometric view key, 738
isosceles triangle shape, 794
isosceles triangle apex angle key, 794
isosceles triangle stretches key, 794
isprime math function, 1039
iterations key, 474

\jobname, 670
join key, 604
jump key, 1213
jump mark left key, 350
jump mark mid key, 350
jump mark right key, 350

key (Lua), 494
key attribute key, 668
Key handlers
    .activate family, 998
    .add, 990
    .add code, 987
    .add style, 988
    .append, 990
    .append code, 988
    .append style, 989
    .belongs to family, 998
    .cd, 986
    .code, 986
    .code 2 args, 987
    .code args, 987
    .code n args, 987
    .deactivate family, 998
    .default, 986
    .ecode, 987
    .ecode 2 args, 987
    .ecode args, 987
    .ecode n args, 987
    .estore in, 990
    .estyle, 988
    .estyle 2 args, 988
    .estyle args, 989
    .evaluated, 991
    .expand once, 991
    .expand twice, 991

.expanded, 991
.forward to, 992
.get, 989
.initial, 989
.install key filter, 997
.install key filter handler, 997
.is choice, 991
.is family, 986, 998
.is if, 990
.lastretry, 994
.link, 990
.list, 992
.pic, 266
.prefix, 990
.prefix code, 988
.prefix style, 989
.retry, 994
.search also, 992
.show code, 995
.show value, 994
.store in, 990
.style, 988
.style 2 args, 988
.style args, 989
.style n args, 988
.style sheet, 933
.tip, 208
.try, 994
.value forbidden, 986
.value required, 986
Kite arrow tip, 217
kite shape, 795
kite lower vertex angle key, 795
kite upper vertex angle key, 795
kite vertex angles key, 796
Koch curve type 1 decoration, 665
Koch curve type 2 decoration, 665
Koch snowflake decoration, 665

l-system key, 701
l-system path operation, 701
label key, 249
label distance key, 250
label position key, 249
large circuit symbols key, 610
large components first key, 435
large mindmap key, 714
.lastretry handler, 994
late options key, 260
LaTeX arrow tip, 217
Latex arrow tip, 217
latitude key, 564
layer distance key, 423
layer post sep key, 423
layer pre sep key, 423
layer sep key, 423
layered graph drawing library, 468
layered layout key, 468
Layout, *see* Page layout
LayoutPipeline (Lua), 520
LayoutPipeline.
anchor (Lua), 522
cutEdges (Lua), 522
decompose (Lua), 522
orient (Lua), 522
packComponents (Lua), 522
prepareBoundingBoxes (Lua), 522
prepareRotateAround (Lua), 522
left key, 197, 238, 241, 849, 1082
left anchor key, 276
left angle key, 699
left color key, 776
left delimiter key, 711
legend key, 946
legend entry options/circular label in legend line key, 962
legend entry options/default label in legend closed path key, 959
legend entry options/default label in legend mark key, 959
legend entry options/default label in legend path key, 959
legend entry options/gap circular label in legend line key, 962
legend entry options/label in legend line coordinates key, 960
legend entry options/label in legend mark coordinates key, 961
legend entry options/label in legend one mark key, 962
legend entry options/label in legend three marks key, 963
legend entry options/legend key, 946
legend entry options/node style key, 956
legend entry options/setup key, 958
legend entry options/straight label in legend line key, 961
legend entry options/text key, 947
legend entry options/text colored key, 956
legend entry options/text left key, 957
legend entry options/text only key, 957
legend entry options/text right key, 957
legend entry options/visualizer in legend key, 958
legend entry options/visualizer in legend style key, 958
legend entry options/zig zag label in legend line key, 961
legend options/above key, 953
legend options/above left of key, 955
legend options/above of key, 955
legend options/above right of key, 955
legend options/anchor key, 951
legend options/at key, 951
legend options/at values key, 955
legend options/below key, 953
legend options/below left of key, 955
legend options/below of key, 955
legend options/below right of key, 955
legend options/columns key, 950
legend options/down then left key, 948
legend options/down then right key, 947
legend options/east inside key, 954
legend options/east outside key, 951
legend options/every legend inside key, 954
legend options/every new legend key, 945
legend options/ideal number of columns key, 950

legend options/ideal number of rows key, 950
legend options/label style key, 956
legend options/left key, 952
legend options/left of key, 955
legend options/left then down key, 948
legend options/left then up key, 948
legend options/matrix node style key, 945
legend options/max columns key, 949
legend options/max rows key, 949
legend options/north east inside key, 954
legend options/north east outside key, 952
legend options/north inside key, 954
legend options/north outside key, 953
legend options/north west inside key, 954
legend options/north west outside key, 952
legend options/opaque key, 954
legend options/right key, 951
legend options/right of key, 955
legend options/right then down key, 948
legend options/right then up key, 948
legend options/rows key, 950
legend options/south east inside key, 953
legend options/south east outside key, 952
legend options/south inside key, 954
legend options/south outside key, 953
legend options/south west inside key, 954
legend options/south west outside key, 953
legend options/transparent key, 954
legend options/up then left key, 948
legend options/up then right key, 948
legend options/west inside key, 954
legend options/west outside key, 952
length key, 192, 449
lens key, 834
less math function, 1042
let path operation, 167
let tikz math function, 704
level key, 307, 334
level 1 concept key, 716
level 2 concept key, 716
level 3 concept key, 716
level 4 concept key, 716
level ⟨*level*⟩ key, 307
level ⟨*number*⟩ key, 334
level distance key, 335, 423
level post sep key, 423
level pre sep key, 423
level sep key, 423
lib (Lua), 534
lib.
    class (Lua), 536
    copy (Lua), 535
    find (Lua), 534
    find_min (Lua), 535
    icopy (Lua), 535
    id (Lua), 536
    imap (Lua), 535
    lookup_option (Lua), 536
    map (Lua), 535
    ondemand (Lua), 536
    random (Lua), 537
    random_permutation (Lua), 535
    randomseed (Lua), 537

Libraries
    3d, 564
    angles, 568
    animations, 380
    arrows.meta, 210
    automata, 571
    babel, 576
    backgrounds, 577
    bending, 202
    calc, 147, 581
    calendar, 582
    chains, 600
    circuits, 609
    circuits.ee, 626
    circuits.ee.IEC, 626
    circuits.logic, 617
    circuits.logic.CDH, 619
    circuits.logic.IEC, 618
    circuits.logic.US, 618
    circular, 481
    curvilinear, 1169
    datavisualization, 855
    datavisualization.formats.functions, 867
    datavisualization.polar, 964
    decorations, 367
    decorations.footprints, 649
    decorations.fractals, 664
    decorations.markings, 644
    decorations.pathmorphing, 638
    decorations.pathreplacing, 641
    decorations.shapes, 651
    decorations.text, 654
    er, 667
    examples, 530
    external, 669
    fadings, 683
    fit, 684
    fixedpointarithmetic, 687
    folding, 723
    force, 473
    fpu, 689
    graphdrawing, 420, 450
    graphs, 267
    graphs.standard, 311
    intersections, 143, 1094
    layered, 468
    lindenmayersystems, 698
    math, 703
    matrix, 709
    mindmap, 713
    patterns, 730
    patterns.meta, 731
    perspective, 738
    petri, 744
    phylogenetics, 483
    plothandlers, 749
    plotmarks, 758
    positioning, 238
    profiler, 760
    quotes, 252
    rdf, 763
    routing, 488
    scopes, 132

shadings, 776
shadows, 780
shapes.arrows, 810
shapes.callouts, 823
shapes.gates.ee, 630
shapes.gates.ee.IEC, 632
shapes.gates.logic, 620
shapes.gates.logic.IEC, 623
shapes.gates.logic.US, 622
shapes.geometric, 786
shapes.misc, 827
shapes.multipart, 816
shapes.symbols, 801
spy, 832
svg.path, 838
through, 844
topaths, 839
trees, 458, 845
turtle, 848
views, 850
light dependent key, 630
light emitting key, 628, 630
lindenmayer system key, 701
lindenmayer system path operation, 701
lindenmayersystems library, 698
line cap key, 174, 199
line join key, 174, 199
line to key, 839
line width animation attribute, 396
line width basic layer animation attribute, 1218
line width key, 173, 200, 734
line width' key, 200
linear key, 1213
linear optimization layer assignment key, 471
linear optimization node positioning key, 472
Lines pattern, 733
lineto decoration, 638
lineto code key, 643
linewidth system layer animation attribute, 1260
.link handler, 990
.list handler, 992
ln math function, 1037
local bounding box key, 1145
log key, 998
log10 math function, 1037
log2 math function, 1037
logic gate anchors use bounding box key, 622
logic gate IEC symbol align key, 624
logic gate IEC symbol color key, 624
logic gate input sep key, 621
logic gate inputs key, 621
logic gate inverted radius key, 621
longitude key, 564
lookup_option (Lua), 536
LookupTable (Lua), 537
LookupTable.
add (Lua), 537
addOne (Lua), 537
remove (Lua), 537
loop key, 842
loop above key, 842
loop below key, 843
loop left key, 843
loop right key, 843
loose background key, 578
loosely dash dot key, 176
loosely dash dot dot key, 176
loosely dashed key, 176
loosely dotted key, 175
looseness key, 841
low key, 909, 914
lower left key, 777
lower right key, 777
lt key, 849

m key, 312
magnetic tape shape, 808
magnetic tape tail key, 809
magnetic tape tail extend key, 808
magnification key, 834
magnifying glass shape, 802
magnifying glass handle angle key, 802
magnifying glass handle aspect key, 802
major key, 894
major also at key, 895
major at key, 895
make contact key, 629
make contact IEC shape, 636
make snapshot after key, 414
make snapshot if necessary key, 414
make snapshot of key, 413
makeRigid (Lua), 516
Mandelbrot set shading, 778
many key, 893
map (Lua), 535
mark key, 346, 645, 647
mark color key, 758
mark connection node key, 648
mark indices key, 347
mark info/distance from start key, 647
mark info/sequence number key, 647
mark options key, 347
mark phase key, 347
mark repeat key, 347
mark size key, 347
markings decoration, 645
matching key, 316
matching and star key, 316
math library, 703
Math functions
abs, 1037
acos, 1041
add, 1035
and, 1042
array, 1045
asin, 1041
atan, 1041
atan2, 1041
bin, 1044
ceil, 1038
cos, 1040
cosec, 1040
cosh, 1045
cot, 1041
deg, 1040
depth, 1046
dim, 1045

div, 1036
divide, 1036
e, 1036
equal, 1042
exp, 1036
factorial, 1036
false, 1043
floor, 1038
frac, 1039
gcd, 1039
greater, 1042
height, 1046
Hex, 1044
hex, 1044
ifthenelse, 1043
int, 1038
iseven, 1039
isodd, 1039
isprime, 1039
less, 1042
ln, 1037
log10, 1037
log2, 1037
max, 1045
min, 1045
Mod, 1037
mod, 1037
multiply, 1035
neg, 1035
not, 1043
notequal, 1042
notgreater, 1042
notless, 1042
oct, 1044
or, 1042
pi, 1039
pow, 1036
rad, 1040
rand, 1043
random, 1043
real, 1039
rnd, 1043
round, 1038
scalar, 1032
sec, 1040
sign, 1037
sin, 1040
sinh, 1045
sqrt, 1036
subtract, 1035
tan, 1040
tanh, 1046
true, 1043
veclen, 1045
width, 1046
math nodes key, 281
Math operators
( ), 1034
*, 1033
||, 1034
+, 1033
-, 1033
/, 1033
<, 1033
<=, 1034
==, 1033
>, 1033
>=, 1033
? :, 1033
" ", 1034
^, 1033
{ }, 1034
!, 1033, 1034
!=, 1033
[ ], 1034
&&, 1034
r, 1033
\matrix, 318
matrix key, 318
matrix library, 709
matrix module, 1152
matrix anchor key, 326
matrix of math nodes key, 710
matrix of nodes key, 709
max math function, 1045
max distance key, 841
medium circuit symbols key, 610
meet key, 850
Mercedes star plot mark, 758
Mercedes star flipped plot mark, 758
meta-amplitude key, 637
meta-segment length key, 637
\method, 1066
Methods
connect, 1071
copy, 1070
emit, 1071
get handle, 1070
get id, 1069
signal, 1071
mid left key, 242
mid right key, 242
middle color key, 776
midway key, 247
min math function, 1045
min distance key, 841
min exponent for 1000 sep key, 1060
mindmap key, 713
mindmap library, 713
minimum coarsening size key, 478
minimum height key, 228, 1140
minimum height layer assignment key, 471
minimum layers key, 469
minimum number of children key, 463
minimum size key, 229, 1140
minimum width key, 228, 1140
minor key, 894
minor also at key, 895
minor at key, 895
minor steps between steps key, 890
mirror key, 370
missing key, 337
missing nodes get space key, 461
miter limit key, 174
Mod math function, 1037
mod math function, 1037

mode key, 676
model (Lua), 498
Modules
    animations, 1209
    decorations, 1108
    matrix, 1152
    nonlineartransformations, 1166
    oo, 1064
    parser, 1024
    plot, 1186
    shapes, 1137
Monday date test, 1010
month code key, 586
month label above centered key, 594
month label above left key, 594
month label above right key, 595
month label below centered key, 595
month label below left key, 595
month label left key, 593
month label left vertical key, 593
month label right key, 593
month label right vertical key, 593
month list key, 592
month text key, 586
month xshift key, 583
month yshift key, 583
motion basic layer animation attribute, 1222
motion system layer animation attribute, 1263
move to key, 839
moveto decoration, 642
moveto code key, 643
moveTowards (Lua), 514
multi key, 286
multiply math function, 1035

\n, 167
n key, 221, 311
nail at key, 433
naive greedy cycle removal key, 471
name key, 131, 139, 144, 153, 224, 279, 358, 368, 701, 731, 1211
name intersections key, 144
name path key, 144
name path global key, 144
name prefix key, 226
name prefix .. key, 265
name separator key, 279
name shore V key, 312
name shore W key, 312
name suffix key, 226
named format, 866
nand gate key, 620
nand gate IEC shape, 625
nand gate IEC symbol key, 624
nand gate US shape, 623
near end key, 247
near start key, 247
nearly opaque key, 354
nearly transparent key, 354
necklace routing key, 488
neg math function, 1035
new (Lua), 498, 501, 505, 511–513, 516, 518–520, 538
new -- key, 277
new -> key, 276
new -!- key, 277
new <- key, 277
new <-> key, 277
new axis base key, 873
new axis system key, 919
new Cartesian axis key, 881
new edge to key, 532
new legend key, 945
new legend entry key, 958
new object key, 863
new polar axes key, 971
new set key, 282, 870
new visualizer key, 928
new_rotation (Lua), 518
new_scaling (Lua), 518
new_shift (Lua), 518
newTableStorage (Lua), 519
next state key, 1112, 1118
no layout key, 449
no markers key, 348
no marks key, 348
no phylogenetic tree optimization key, 485
no placement key, 299
no span edge key, 466
no tick text key, 915
no tick text at key, 900
Node, *see* Predefined node
\node, 226
node coordinate system, 139
node key, 141
node path operation, 222
node also path operation, 260
node contents key, 223
node distance key, 240, 422
node font key, 231
node halign header key, 235
node post sep key, 423
node pre sep key, 422
node quotes mean key, 254
node sep key, 423
node style key, 896
node styling key, 897
\nodepart, 230
\nodeparts, 1146
nodes key, 273, 322
nodes behind edges key, 448
nodes in empty cells key, 710
none key, 894
nonlineartransformations module, 1166
nonzero rule key, 180
nor gate key, 620
nor gate IEC symbol key, 624
norm (Lua), 514
normalize (Lua), 514
normalized (Lua), 515
north fading, 683
north east lines pattern, 730
north west lines pattern, 730
not key, 1000
not math function, 1043
not ⟨*color class name*⟩ key, 292
not gate key, 620
not gate IEC symbol key, 624

notequal math function, 1042
notgreater math function, 1042
notless math function, 1042
nudge key, 432
nudge down key, 432
nudge left key, 432
nudge right key, 432
nudge up key, 432
number nodes key, 279
number nodes sep key, 279
numbered faces key, 725

o plot mark, 758
object class, 1070
object key, 384
oct math function, 1044
octahedron folding pic type, 725, 726
of key, 144
ohm key, 627, 630
ohmmeter key, 629
on background layer key, 577
on chain key, 602
on grid key, 240
ondemand (Lua), 536
only marks key, 352
only named key, 678
oo module, 1064
opacity animation attribute, 395
opacity basic layer animation attribute, 1217
opacity key, 353
opacity system layer animation attribute, 1257
opaque key, 354
open key, 198
operator key, 293
oplus plot mark, 758
oplus* plot mark, 758
optimize key, 677
optimize command away key, 677
optimize/install key, 678
optimize/restore key, 678
options (Lua), 508
options at key, 900
Options for graphics, *see* Graphic options and styles
Options for packages, *see* Package options
options/along key, 400
options/begin key, 405
options/begin on key, 405
options/begin snapshot key, 413
options/ease key, 411
options/ease in key, 411
options/ease out key, 411
options/end key, 405
options/end on key, 409
options/entry control key, 411
options/exit control key, 410
options/forever key, 405
options/freeze key, 405
options/jump key, 412
options/origin key, 401
options/repeat key, 410
options/repeats key, 409
options/restart key, 408
options/stay key, 412
options/transform key, 402
optionsAccumulated (Lua), 508
optionsArray (Lua), 507
or key, 1000
or math function, 1042
or gate key, 620
or gate IEC symbol key, 624
order key, 701
orderIncoming (Lua), 502
orderOutgoing (Lua), 502
orient (Lua), 522
orient key, 428
orient head key, 429
orient tail key, 429
orient' key, 429
origin key, 1222
otimes plot mark, 758
otimes* plot mark, 758
out key, 839
out control key, 842
out distance key, 842
out looseness key, 841
out max distance key, 842
out min distance key, 841
outer color key, 779
outer frame sep key, 579
outer frame xsep key, 579
outer frame ysep key, 579
outer sep key, 227, 1141
outer xsep key, 228, 1140
outer ysep key, 228, 1141
outgoing (Lua), 502
outlier key, 926
output format key, 689
overlay key, 258

\p, 167
p key, 739
Package options for PGF
    draft, 1076, 1175
    version=⟨*version*⟩, 1076
Packages and files
    pgf, 1076
    pgf.cfg, 1234
    pgfcalendar, 1009
    pgfcore, 1076
    pgfexternal.tex, 1181
    pgffor, 1003
    pgfkeys, 975
    pgfmath, 1029
    pgfparser, 1024
    pgfsys, 1234
    pgfsys-common-pdf, 1234
    pgfsys-common-postscript, 1234
    pgfsys-dvi.def, 122
    pgfsys-dvipdfm.def, 120
    pgfsys-dvips.def, 120
    pgfsys-dvisvgm.def, 120
    pgfsys-pdftex.def, 119
    pgfsys-tex4ht.def, 121
    pgfsys-xetex.def, 120
    tikz, 128
packComponents (Lua), 522
pad (Lua), 517
padded key, 910

Page layouts
16 on 1, 1018
2 on 1, 1018
4 on 1, 1018
6 on 1, 1018
8 on 1, 1018
resize to, 1017
rounded corners, 1018
two screens with lagging second, 1019
two screens with optional second, 1019
parabola path operation, 161
parabola height key, 162
parameters key, 732
parametric key, 345
parent anchor key, 340
Parenthesis arrow tip, 213
parse key, 275, 1007
parser module, 1024
part distance key, 424
part post sep key, 424
part pre sep key, 424
part sep key, 424
Path (Lua), 515
\path, 152
path animation attribute, 397
path basic layer animation attribute, 1219
path key, 315
path system layer animation attribute, 1258
path fading key, 360
path has corners key, 637
Path operations
--, 154
-|, 155
|-, 155
--plot, 341
.., 155
:, 169
arc, 158
child, 331
circle, 157
coordinate, 226
cos, 162
decorate, 367
edge, 256
edge from parent, 338
ellipse, 158
⟨*empty*⟩, 154
foreach, 166
graph, 272
grid, 159
l-system, 701
let, 167
lindenmayer system, 701
node, 222
node also, 260
parabola, 161
pic, 261
plot, 341
rectangle, 156
sin, 162
svg, 163
to, 163
path picture key, 181
path picture bounding box node, 181
Path.
new (Lua), 516
Path:
appendArc (Lua), 517
appendArcTo (Lua), 518
appendClosepath (Lua), 516
appendCurveto (Lua), 516
appendLineto (Lua), 516
appendMoveto (Lua), 516
boundingBox (Lua), 517
clear (Lua), 516
clone (Lua), 516
coordinates (Lua), 517
cutAtBeginning (Lua), 517
cutAtEnd (Lua), 517
intersectionsWith (Lua), 517
makeRigid (Lua), 516
pad (Lua), 517
reversed (Lua), 516
shift (Lua), 516
shiftByCoordinate (Lua), 516
transform (Lua), 516
\pattern, 171
pattern key, 179
pattern color key, 179
Patterns
bricks, 730
checkerboard, 730
checkerboard light gray, 731
crosshatch, 730
crosshatch dots, 730
crosshatch dots gray, 731
crosshatch dots light steel blue, 731
Dots, 735
dots, 730
fivepointed stars, 730
grid, 730
Hatch, 734
horizontal lines, 730
horizontal lines dark blue, 731
horizontal lines dark gray, 731
horizontal lines gray, 731
horizontal lines light blue, 731
horizontal lines light gray, 731
Lines, 733
north east lines, 730
north west lines, 730
sixpointed stars, 730
Stars, 735
vertical lines, 730
patterns library, 730
patterns.meta library, 731
pentagon plot mark, 758
pentagon* plot mark, 758
perpendicular coordinate system, 143
persistent postcomputation key, 1112
persistent precomputation key, 1112
perspective key, 739
perspective library, 738
petri library, 744
/pgf/
and gate IEC symbol, 624

animate/
  events/click, 406
animation/
  along, 1223
  arrows, 1219
  begin, 1225
  begin on, 1225
  end, 1225
  end on, 1226
  entry, 1212
  events/begin, 407
  events/delay, 408
  events/end, 407
  events/event, 406
  events/focus in, 408
  events/focus out, 408
  events/key, 408
  events/mouse down, 406
  events/mouse move, 407
  events/mouse out, 407
  events/mouse over, 407
  events/mouse up, 407
  events/of, 406
  events/of next, 406
  events/repeat, 408
  freeze at end, 1225
  name, 1211
  origin, 1222
  repeat, 1227
  repeats, 1226
  restart, 1226
  rotate along, 1223
  shorten <, 1220
  shorten >, 1220
  whom, 1211
animations/
  entry control, 1212
  exit control, 1212
  jump, 1213
  linear, 1213
  stay, 1213
arrow box arrows, 814
arrow box east arrow, 814
arrow box head extend, 814
arrow box head indent, 814
arrow box north arrow, 814
arrow box shaft width, 814
arrow box south arrow, 814
arrow box tip angle, 814
arrow box west arrow, 814
arrow keys/
  angle, 194
  angle', 194
  arc, 195
  bend, 204
  cap angle, 221
  color, 197
  fill, 198
  flex, 202
  flex', 203
  harpoon, 196
  inset, 194
  inset', 194
  left, 197
  length, 192
  line cap, 199
  line join, 199
  line width, 200
  line width', 200
  n, 221
  open, 198
  quick, 201
  reversed, 196
  right, 197
  round, 200
  scale, 195
  scale length, 195
  scale width, 195
  sep, 206
  sharp, 200
  slant, 196
  swap, 197
  width, 193
  width', 193
aspect, 787, 800
bar interval shift, 754
bar interval width, 754
bar shift, 753
bar width, 753
buffer gate IEC symbol, 624
callout absolute pointer, 824
callout pointer arc, 825
callout pointer end size, 827
callout pointer segments, 827
callout pointer shorten, 824
callout pointer start size, 826
callout pointer width, 825
callout relative pointer, 824
chamfered rectangle angle, 830
chamfered rectangle corners, 831
chamfered rectangle sep, 831
chamfered rectangle xsep, 830
chamfered rectangle ysep, 831
circular sector angle, 799
cloud ignores aspect, 802
cloud puff arc, 802
cloud puffs, 802
cylinder body fill, 800
cylinder end fill, 800
cylinder uses custom fill, 800
dart tail angle, 797
dart tip angle, 797
data/
  format, 859, 869
  headline, 866
  inline, 869
  new set, 870
  read from file, 859, 869
  samples, 867
  separator, 866
  set, 923
  store in set, 871
  use set, 871
data visualization/
  style sheets/⟨*style sheet*⟩/default style, 934

declare function/
execute at begin function, 1052
execute at end function, 1052
ignore spaces, 1052
declare function, 1051
decoration/
amplitude, 637
anchor, 652
angle, 637
aspect, 637
closepath code, 644
curveto code, 644
end radius, 637
foot angle, 650
foot length, 650
foot of, 650
foot sep, 650
lineto code, 643
mark, 645, 647
mark connection node, 648
mark info/distance from start, 647
mark info/sequence number, 647
meta-amplitude, 637
meta-segment length, 637
mirror, 370
moveto code, 643
name, 368
path has corners, 637
pre, 371
pre length, 371
radius, 637
raise, 370
reset marks, 648
reverse path, 656
segment length, 637
shape, 652
shape end height, 654
shape end size, 654
shape end width, 654
shape evenly spread, 653
shape height, 651
shape scaled, 654
shape sep, 653
shape size, 651
shape sloped, 653
shape start height, 654
shape start size, 654
shape start width, 654
shape width, 651
start radius, 637
stride length, 650
text, 655, 657
text align, 656, 658
text align/align, 656
text align/center, 656
text align/fit to path, 657
text align/fit to path stretching spaces, 657
text align/left, 656
text align/left indent, 656
text align/right, 656
text align/right indent, 656
text color, 656
text effects/character ⟨*number*⟩, 659
text effects/character command, 664
text effects/character count, 660
text effects/character total, 660
text effects/character widths, 659
text effects/characters, 659
text effects/every character, 658
text effects/every character width, 659
text effects/every first letter, 659
text effects/every last letter, 659
text effects/every letter, 659
text effects/every word, 659
text effects/every word separator, 659
text effects/fit text to path, 662
text effects/group letters, 663
text effects/letter ⟨*number*⟩, 659
text effects/letter count, 660
text effects/path from text, 661
text effects/path from text angle, 662
text effects/repeat text, 664
text effects/replace characters, 664
text effects/reverse text, 663
text effects/scale text to path, 662
text effects/style characters, 661
text effects/text along path, 658
text effects/word ⟨*m*⟩ letter ⟨*n*⟩, 659
text effects/word ⟨*number*⟩, 659
text effects/word count, 661
text effects/word separator, 659
text effects/word total, 661
text format delimiters, 656
text/effetcs/letter total, 661
transform, 370
decoration, 367
decoration automaton/
auto corner on length, 1112
auto end on length, 1112
if input segment is closepath, 1112
next state, 1112
persistent postcomputation, 1112
persistent precomputation, 1112
repeat state, 1112
switch if input segment less than, 1111
switch if less than, 1111
width, 1111
decorations/
post, 371
post length, 372
direction ee arrow, 631
double arrow head extend, 812
double arrow head indent, 812
double arrow tip angle, 812
every data, 870
every decoration, 1116
fixed point/
scale file plot x, 688
scale file plot y, 688
scale file plot z, 688
scale results, 687
fixed point arithmetic, 687
foreach/
count, 1007
evaluate, 1006

  expand list, 1007
  parse, 1007
  remember, 1007
  var, 1006
fpu/
  handlers/empty number, 692
  handlers/invalid number, 692
  handlers/wrong lowlevel format, 692
  install only, 690
  output format, 689
  rel thresh, 696
  scale file plot x, 690
  scale file plot y, 690
  scale file plot z, 690
  scale results, 690
fpu, 689
gap around stream point, 755
generic circle IEC/
  before background, 632
generic diode IEC/
  before background, 633
handle new data sets in plots, 1187
handle outlier points in plots, 1187
handle undefined points in plots, 1187
images/
  external info, 681
  include external, 678, 1182
inner sep, 227, 1140
inner xsep, 227, 1140
inner ysep, 227, 1140
isosceles triangle apex angle, 794
isosceles triangle stretches, 794
key filter handlers/
  append filtered to, 998
  ignore, 998
  log, 998
key filters/
  active families, 999
  active families and known, 999
  active families or descendants of, 999
  active families or no family, 999
  active families or no family DEBUG, 999
  and, 1000
  defined, 1000
  equals, 1000
  false, 1000
  is descendant of, 1000
  not, 1000
  or, 1000
  true, 1000
kite lower vertex angle, 795
kite upper vertex angle, 795
kite vertex angles, 796
l-system, 701
lindenmayer system/
  anchor, 702
  axiom, 701
  left angle, 699
  name, 701
  order, 701
  randomize angle percent, 699
  randomize step percent, 699
  right angle, 699
  rule set, 702
  step, 699
lindenmayer system, 701
local bounding box, 1145
logic gate anchors use bounding box, 622
logic gate IEC symbol align, 624
logic gate IEC symbol color, 624
logic gate input sep, 621
logic gate inputs, 621
logic gate inverted radius, 621
magnetic tape tail, 809
magnetic tape tail extend, 808
magnifying glass handle angle, 802
magnifying glass handle aspect, 802
mark color, 758
meta-decoration automaton/
  next state, 1118
  switch if less than, 1118
  width, 1118
minimum height, 228, 1140
minimum size, 229, 1140
minimum width, 228, 1140
nand gate IEC symbol, 624
nor gate IEC symbol, 624
not gate IEC symbol, 624
number format/
  @dec sep mark, 1062
  @sci exponent mark, 1062
  1000 sep, 1059
  1000 sep in fractionals, 1059
  assume math mode, 1062
  dec sep, 1058
  every relative, 1055
  fixed, 1053
  fixed relative, 1056
  fixed zerofill, 1053
  frac, 1057
  frac denom, 1057
  frac shift, 1058
  frac TeX, 1057
  frac whole, 1057
  int detect, 1056
  int trunc, 1057
  min exponent for 1000 sep, 1060
  precision, 1058
  print sign, 1061
  read comma as period, 1058
  relative style, 1056
  relative*, 1054
  retain unit mantissa, 1062
  sci, 1053
  sci 10^e, 1061
  sci 10e, 1061
  sci E, 1061
  sci e, 1061
  sci generic, 1061
  sci generic/exponent, 1062
  sci generic/mantissa sep, 1062
  sci precision, 1058
  sci subscript, 1061
  sci superscript, 1061
  sci zerofill, 1054
  set decimal separator, 1058

set thousands separator, 1059
showpos, 1061
skip 0., 1060
std, 1054
use comma, 1060
use period, 1060
verbatim, 1062
zerofill, 1054
or gate IEC symbol, 624
outer sep, 227, 1141
outer xsep, 228, 1140
outer ysep, 228, 1141
pattern keys/
angle, 733–735
distance, 733–735
line width, 734
points, 735
radius, 735
xshift, 733–735
yshift, 733–735
patterns/
bottom left, 732
code, 732
defaults, 732
name, 731
parameters, 732
set up code, 732
tile size, 732
tile transformation, 732
top right, 732
type, 731
x, 731
y, 731
plot/
gnuplot call, 1190
random starburst, 804
rectangle split allocate boxes, 821
rectangle split draw splits, 822
rectangle split empty part depth, 821
rectangle split empty part height, 821
rectangle split empty part width, 821
rectangle split horizontal, 821
rectangle split ignore empty parts, 821
rectangle split part align, 821
rectangle split part fill, 822
rectangle split parts, 821
rectangle split use custom fill, 822
regular polygon sides, 791
rounded rectangle arc length, 829
rounded rectangle east arc, 829
rounded rectangle left arc, 829
rounded rectangle right arc, 829
rounded rectangle west arc, 829
shape aspect, 229
shape border rotate, 230
shape border uses incircle, 230
signal from, 806
signal pointer angle, 806
signal to, 806
single arrow head extend, 810
single arrow head indent, 811
single arrow tip angle, 810
star point height, 793
star point ratio, 793
star points, 793
starburst point height, 804
starburst points, 804
step, 1103
stepx, 1103
stepy, 1103
tape bend bottom, 807
tape bend height, 807
tape bend top, 807
tex4ht node/
class, 122
css, 122
escape, 121
id, 122
text/
at, 1083
base, 1083
bottom, 1083
left, 1082
right, 1082
rotate, 1084
top, 1082
x, 1083
y, 1083
text mark, 758
text mark as node, 759
text mark style, 759
tips, 190
trapezium angle, 788
trapezium left angle, 788
trapezium right angle, 788
trapezium stretches, 789
trapezium stretches body, 789
trig format, 1041
trim lowlevel, 185
xnor gate IEC symbol, 624
xor gate IEC symbol, 624
pgf package, 1076
pgf (Lua), 534
pgf.
debug (Lua), 534
pgf.cfg file, 1234
pgf.gd.
model (Lua), 498
\pgf@pathmaxx, 1106
\pgf@pathmaxy, 1106
\pgf@pathminx, 1106
\pgf@pathminy, 1106
\pgf@picmaxx, 1107
\pgf@picmaxy, 1107
\pgf@picminx, 1107
\pgf@picminy, 1107
\pgf@process, 1096
\pgf@protocolsizes, 1107
\pgf@relevantforpicturesizefalse, 1107
\pgf@relevantforpicturesizetrue, 1107
\pgf@sys@bp, 1250
\pgf@x, 1228
\pgf@xa, 1228
\pgf@xb, 1228
\pgf@xc, 1228
\pgf@y, 1228

\pgf@ya, 1228
\pgf@yb, 1228
\pgf@yc, 1228
\pgfactualjobname, 670
\pgfaliasid, 1086
\pgfaliasimage, 1176
\pgfalternateextension, 1176
\pgfanimateattribute, 1209
\pgfanimateattributecode, 1211
\pgfapproximatenonlineartransformation, 1169
\pgfapproximatenonlineartranslation, 1169
\pgfarrowsaddtolateoptions, 1136
\pgfarrowsaddtolengthscalelist, 1136
\pgfarrowsaddtooptions, 1135
\pgfarrowsaddtowidthscalelist, 1136
\pgfarrowshullpoint, 1131
\pgfarrowslengthdependent, 1136
\pgfarrowslinewidthdependent, 1136
\pgfarrowssave, 1131
\pgfarrowssavethe, 1131
\pgfarrowssetbackend, 1130
\pgfarrowssetlineend, 1130
\pgfarrowssettipend, 1130
\pgfarrowssetvisualbackend, 1130
\pgfarrowssetvisualtipend, 1130
\pgfarrowsthreeparameters, 1136
\pgfarrowsupperhullpoint, 1131
\pgfcalendar, 1012
pgfcalendar package, 1009
\pgfcalendardatetojulian, 1009
\pgfcalendareastersunday, 1010
\pgfcalendarifdate, 1010
\pgfcalendarjuliantodate, 1010
\pgfcalendarjuliantoweekday, 1010
\pgfcalendarmonthname, 1012
\pgfcalendarmonthshortname, 1012
\pgfcalendarshorthand, 1014
\pgfcalendarsuggestedname, 1015
\pgfcalendarweekdayname, 1011
\pgfcalendarweekdayshortname, 1012
\pgfclearid, 1085
\pgfcoordinate, 1139
pgfcore package, 1076
\pgfcurvilineardistancetotime, 1170
\pgfdata, 869
\pgfdeclarearrow, 1129
\pgfdeclaredataformat, 871
\pgfdeclaredecoration, 1110
\pgfdeclarefading, 1205
\pgfdeclarefunctionalshading, 1197
\pgfdeclarehorizontalshading, 1196
\pgfdeclareimage, 1175
\pgfdeclarelayer, 1193
\pgfdeclarelindenmayersystem, 698
\pgfdeclaremask, 1177
\pgfdeclaremetadecorate, 1118
\pgfdeclarepattern, 731
\pgfdeclarepatternformonly, 1172
\pgfdeclarepatterninherentlycolored, 1173
\pgfdeclareplothandler, 1191
\pgfdeclareplotmark, 757
\pgfdeclareradialshading, 1196
\pgfdeclareshape, 1146
\pgfdeclareverticalshading, 1196
\pgfdecorateaftercode, 1117
\pgfdecoratebeforecode, 1117
\pgfdecoratecurrentpath, 1117
\pgfdecoratedangle, 1113
\pgfdecoratedcompleteddistance, 1113
\pgfdecoratedinputsegmentcompleteddistance, 1113
\pgfdecoratedinputsegmentlength, 1113
\pgfdecoratedinputsegmentremainingdistance, 1113
\pgfdecoratedpath, 1116
\pgfdecoratedpathlength, 1112
\pgfdecoratedremainingdistance, 1113
\pgfdecorateexistingpath, 1116
\pgfdecoratepath, 1117
pgfdecoration environment, 1114, 1117
\pgfdecorationpath, 1116
\pgfdvdeclarestylesheet, 935
\pgferror, 1087
pgfexternal.tex file, 1181
\pgfextra, 169
\pgfextractx, 1095
\pgfextracty, 1095
pgffor package, 1003
\pgffuncshadingcmyktogray, 1200
\pgffuncshadingcmyktorgb, 1200
\pgffuncshadinggraytocmyk, 1200
\pgffuncshadinggraytorgb, 1200
\pgffuncshadingrgbtocmyk, 1200
\pgffuncshadingrgbtogray, 1200
\pgfgaliasid, 1086
\pgfgdaddspecificationhook, 451
\pgfgdbegineventgroup, 456
\pgfgdbeginlayout, 454
\pgfgdbeginscope, 451
\pgfgdedge, 452
\pgfgdendeventgroup, 456
\pgfgdendlayout, 454
\pgfgdendscope, 452
\pgfgdevent, 456
\pgfgdeventgroup, 456
\pgfgdsetedgecallback, 453
\pgfgdsetlatenodeoption, 454
\pgfgdsetrequestcallback, 455
\pgfgdsubgraphnode, 457
\pgfgetlastxy, 1095
\pgfgettransform, 1162
\pgfgettransformentries, 1162
\pgfhorizontaltransformationadjustment, 1163
\pgfidrefnextuse, 1086
\pgfidrefprevuse, 1086
pgfidscope environment, 1084
\pgfifidreferenced, 1086
\pgfimage, 1177
pgfinterruptboundingbox environment, 1082
pgfinterruptpath environment, 1081
pgfinterruptpicture environment, 1081
\pgfintersectionofpaths, 1094
\pgfintersectionsolutions, 1094
\pgfintersectionsortbyfirstpath, 1095
\pgfintersectionsortbysecondpath, 1095
\pgfkeys, 979

pgfkeys package, 975
pgfkeys Libraries
filtered, 995
\pgfkeysactivatefamilies, 1001
\pgfkeysactivatefamiliesandfilteroptions, 999
\pgfkeysactivatefamily, 1001
\pgfkeysactivatesinglefamilyandfilteroptions, 999
\pgfkeysalso, 979
\pgfkeysalsofiltered, 997
\pgfkeysalsofilteredfrom, 997
\pgfkeysalsofrom, 997
\pgfkeysdeactivatefamily, 1001
\pgfkeysdef, 982
\pgfkeysdefargs, 982
\pgfkeysdefnargs, 982
\pgfkeysdetokenizedvalueof, 978
\pgfkeysedef, 982
\pgfkeysedefargs, 982
\pgfkeysedefnargs, 982
\pgfkeysevalkeyfilterwith, 1002
\pgfkeysfiltered, 996
\pgfkeysgetfamily, 1001
\pgfkeysgetvalue, 977
\pgfkeysifassignable, 978
\pgfkeysifdefined, 978
\pgfkeysifempty, 978
\pgfkeysiffamilydefined, 1001
\pgfkeysinstallkeyfilter, 1001
\pgfkeysinstallkeyfilterhandler, 1001
pgfkeysinterruptkeyfilter environment, 1001
\pgfkeysisfamilyactive, 1001
\pgfkeyslet, 977
\pgfkeyssavekeyfilterstateto, 1001
\pgfkeyssetevalue, 977
\pgfkeyssetfamily, 1001
\pgfkeyssetvalue, 977
\pgfkeysunexpandedvalueof, 977
\pgfkeysvalueof, 977
\pgflibraryfpuifactive, 690
\pgflindenmayersystem, 701
\pgflinewidth, 1122
\pgflowlevel, 1164
\pgflowlevelobj, 1164
pgflowlevelscope environment, 1165
\pgflowlevelsynccm, 1164
\pgflsystemcurrentleftangle, 699
\pgflsystemcurrentrightangle, 699
\pgflsystemcurrentstep, 699
\pgflsystemdrawforward, 700
\pgflsystemmoveforward, 700
\pgflsystemrandomizeleftangle, 700
\pgflsystemrandomizerightangle, 700
\pgflsystemrandomizestep, 699
\pgflsystemrestorestate, 700
\pgflsystemsavestate, 700
\pgflsystemturnleft, 700
\pgflsystemturnright, 700
pgfmath package, 1029
\pgfmathabs, 1037
\pgfmathacos, 1041
\pgfmathadd, 1035
\pgfmathaddtocount, 1031
\pgfmathaddtocounter, 1032
\pgfmathaddtolength, 1031
\pgfmathand, 1042
\pgfmathanglebetweenlines, 1049
\pgfmathanglebetweenpoints, 1049
\pgfmathapproxequalto, 1047
\pgfmatharray, 1045
\pgfmathasin, 1041
\pgfmathatan, 1041
\pgfmathatantwo, 1041
\pgfmathbasetoBase, 1048
\pgfmathbasetobase, 1048
\pgfmathbasetodec, 1048
\pgfmathbin, 1044
\pgfmathceil, 1038
\pgfmathcos, 1040
\pgfmathcosec, 1040
\pgfmathcosh, 1045
\pgfmathcot, 1041
\pgfmathdeclarefunction, 1050
\pgfmathdeclarerandomlist, 1047
\pgfmathdectoBase, 1048
\pgfmathdectobase, 1048
\pgfmathdeg, 1040
\pgfmathdepth, 1046
\pgfmathdim, 1045
\pgfmathdiv, 1036
\pgfmathdivide, 1036
\pgfmathe, 1036
\pgfmathequal, 1042
\pgfmathexp, 1036
\pgfmathfactorial, 1036
\pgfmathfalse, 1043
\pgfmathfloat, 695
\pgfmathfloatabserror, 696
\pgfmathfloatcreate, 693
\pgfmathfloatgetexponent, 694
\pgfmathfloatgetflags, 693
\pgfmathfloatgetflagstomacro, 694
\pgfmathfloatgetmantissa, 694
\pgfmathfloatgetmantissatok, 694
\pgfmathfloatifapproxequalrel, 696
\pgfmathfloatifflags, 693
\pgfmathfloatint, 696
\pgfmathfloatlessthan, 695
\pgfmathfloatmultiplyfixed, 696
\pgfmathfloatparsenumber, 691
\pgfmathfloatqparsenumber, 692
\pgfmathfloatrelerror, 696
\pgfmathfloatround, 694
\pgfmathfloatroundzerofill, 695
\pgfmathfloatsetextprecision, 695
\pgfmathfloatshift, 696
\pgfmathfloattoextentedprecision, 695
\pgfmathfloattofixed, 692
\pgfmathfloattoint, 692
\pgfmathfloattomacro, 693
\pgfmathfloattoregisters, 693
\pgfmathfloattoregisterstok, 693
\pgfmathfloattosci, 692
\pgfmathfloatvalueof, 693
\pgfmathfloor, 1038
\pgfmathfrac, 1039

\pgfmathgcd, 1039
\pgfmathgeneratepseudorandomnumber, 1047
\pgfmathgreater, 1042
\pgfmathheight, 1046
\pgfmathHex, 1044
\pgfmathhex, 1044
\pgfmathifisint, 1056
\pgfmathifthenelse, 1043
\pgfmathint, 1038
\pgfmathiseven, 1039
\pgfmathisodd, 1039
\pgfmathisprime, 1039
\pgfmathless, 1042
\pgfmathln, 1037
\pgfmathlog, 696
\pgfmathlogten, 1037
\pgfmathlogtwo, 1037
\pgfmathmax, 1045
\pgfmathmin, 1045
\pgfmathMod, 1037
\pgfmathmod, 1037
\pgfmathmultiply, 1035
\pgfmathneg, 1035
\pgfmathnot, 1043
\pgfmathnotequal, 1042
\pgfmathnotgreater, 1042
\pgfmathnotless, 1042
\pgfmathoct, 1044
\pgfmathor, 1043
\pgfmathparse, 1030
\pgfmathpi, 1039
\pgfmathpostparse, 1031
\pgfmathpow, 1036
\pgfmathprintnumber, 1053
\pgfmathprintnumberto, 1053
\pgfmathqparse, 1030
\pgfmathrad, 1040
\pgfmathrand, 1043
\pgfmathrandom, 1044
\pgfmathrandominteger, 1047
\pgfmathrandomitem, 1047
\pgfmathreal, 1039
\pgfmathreciprocal, 1047
\pgfmathredeclarefunction, 1051
\pgfmathrnd, 1043
\pgfmathround, 1038
\pgfmathroundto, 694
\pgfmathroundtozerofill, 694
\pgfmathscalar, 1032
\pgfmathsec, 1040
\pgfmathsetbasenumberlength, 1048
\pgfmathsetcount, 1031
\pgfmathsetcounter, 1032
\pgfmathsetlength, 1031
\pgfmathsetlengthmacro, 1032
\pgfmathsetmacro, 1032
\pgfmathsetseed, 1048
\pgfmathsign, 1038
\pgfmathsin, 1040
\pgfmathsinh, 1045
\pgfmathsqrt, 1036
\pgfmathsubtract, 1035
\pgfmathtan, 1040
\pgfmathtanh, 1046
\pgfmathtodigitlist, 1049
\pgfmathtrue, 1043
\pgfmathtruncatemacro, 1032
\pgfmathveclen, 1045
\pgfmathwidth, 1046
\pgfmatrix, 1152
\pgfmatrixbegincode, 1156
\pgfmatrixcurrentcolumn, 1156
\pgfmatrixcurrentrow, 1156
\pgfmatrixemptycode, 1156
\pgfmatrixendcode, 1156
\pgfmatrixendrow, 1155
\pgfmatrixnextcell, 1154
\pgfmetadecoratedcompleteddistance, 1119
\pgfmetadecoratedinputsegmentcompleteddistance, 1119
\pgfmetadecoratedinputsegmentremainingdistance, 1119
\pgfmetadecoratedpathlength, 1118
\pgfmetadecoratedremainingdistance, 1119
pgfmetadecoration environment, 1120
\pgfmultipartnode, 1139
\pgfnode, 1138
\pgfnodealias, 1140
\pgfnodepostsetupcode, 1142
\pgfnoderename, 1140
pgfonlayer environment, 1193, 1194
\pgfooappend, 1068
\pgfooclass, 1064
\pgfooeset, 1068
\pgfoogc, 1066
\pgfooget, 1069
\pgfoolet, 1068
\pgfoonew, 1065
\pgfooobj, 1069
\pgfooprefix, 1068
\pgfooset, 1068
\pgfoosuper, 1067
\pgfoothis, 1067
\pgfoovalueof, 1068
\pgfpagescurrentpagewillbelogicalpage, 1022
\pgfpagesdeclarelayout, 1019
\pgfpageslogicalpageoptions, 1021
\pgfpagesphysicalpageoptions, 1020
\pgfpagesshipoutlogicalpage, 1022
\pgfpagesuselayout, 1017
/pgfparser/
  ⟨*parser name*⟩/
    silent, 1026
  silent, 1026
  status, 1026
pgfparser package, 1024
\pgfparserdef, 1024
\pgfparserdeffinal, 1025
\pgfparserdefunknown, 1025
\pgfparserifmark, 1026
\pgfparserlet, 1025
\pgfparserletter, 1026
\pgfparserparse, 1024
\pgfparserreinsert, 1026
\pgfparserset, 1026
\pgfparserstate, 1026

\pgfparserswitch, 1025
\pgfparsertoken, 1026
\pgfparsetime, 1214
\pgfpatharc, 1100
\pgfpatharcaxes, 1100
\pgfpatharcto, 1101
\pgfpatharctomaxstepsize, 1102
\pgfpatharctoprecomputed, 1101
\pgfpathcircle, 1103
\pgfpathclose, 1099
\pgfpathcosine, 1105
\pgfpathcurvebetweentime, 1099
\pgfpathcurvebetweentimecontinue, 1099
\pgfpathcurveto, 1098
\pgfpathellipse, 1102
\pgfpathgrid, 1103
\pgfpathlineto, 1098
\pgfpathmoveto, 1097
\pgfpathparabola, 1104
\pgfpathqcircle, 1231
\pgfpathqcurveto, 1230
\pgfpathqlineto, 1230
\pgfpathqmoveto, 1230
\pgfpathquadraticcurveto, 1099
\pgfpathrectangle, 1103
\pgfpathrectanglecorners, 1103
\pgfpathsine, 1104
\pgfpathsvg, 838
pgfpicture environment, 1077, 1079
\pgfplotbarwidth, 753
\pgfplotfunction, 1189
\pgfplotgnuplot, 1189
\pgfplothandlerclosedcurve, 749
\pgfplothandlerconstantlineto, 750
\pgfplothandlerconstantlinetomarkmid, 751
\pgfplothandlerconstantlinetomarkright, 750
\pgfplothandlercurveto, 749
\pgfplothandlerdiscard, 1191
\pgfplothandlergapcycle, 755
\pgfplothandlergaplineto, 755
\pgfplothandlerjumpmarkleft, 751
\pgfplothandlerjumpmarkmid, 751
\pgfplothandlerjumpmarkright, 751
\pgfplothandlerlineto, 1190
\pgfplothandlermark, 755
\pgfplothandlermarklisted, 756
\pgfplothandlermonotone, 750
\pgfplothandlerpolarcomb, 752
\pgfplothandlerpolygon, 1191
\pgfplothandlerrecord, 1191
\pgfplothandlerxbar, 753
\pgfplothandlerxbarinterval, 754
\pgfplothandlerxcomb, 752
\pgfplothandlerybar, 753
\pgfplothandlerybarinterval, 753
\pgfplothandlerycomb, 752
\pgfplotmarksize, 757
\pgfplotstreamend, 1188
\pgfplotstreamnewdataset, 1187
\pgfplotstreampoint, 1187
\pgfplotstreampointoutlier, 1187
\pgfplotstreampointundefined, 1187
\pgfplotstreamspecial, 1188
\pgfplotstreamstart, 1186
\pgfplotxyfile, 1188
\pgfplotxyzfile, 1188
\pgfplotxzerolevelstreamconstant, 753
\pgfplotyzerolevelstreamconstant, 753
\pgfpoint, 1088
\pgfpointadd, 1090
\pgfpointanchor, 1143
\pgfpointarcaxesattime, 1092
\pgfpointborderellipse, 1093
\pgfpointborderrectangle, 1093
\pgfpointcurveattime, 1092
\pgfpointcurvilinearbezierorthogonal, 1170
\pgfpointcurvilinearbezierpolar, 1171
\pgfpointcylindrical, 1090
\pgfpointdecoratedinputsegmentlast, 1113
\pgfpointdecoratedpathfirst, 1116
\pgfpointdecoratedpathlast, 1113, 1116
\pgfpointdecorationpathlast, 1116
\pgfpointdiff, 1091
\pgfpointintersectionofcircles, 1094
\pgfpointintersectionoflines, 1093
\pgfpointintersectionsolution, 1094
\pgfpointlineatdistance, 1092
\pgfpointlineattime, 1091
\pgfpointmetadecoratedpathfirst, 1118
\pgfpointmetadecoratedpathlast, 1118
\pgfpointnormalised, 1091
\pgfpointorigin, 1088
\pgfpointpolar, 1088
\pgfpointpolarxy, 1089
\pgfpointscale, 1091
\pgfpointshapeborder, 1143
\pgfpointspherical, 1090
\pgfpointtransformed, 1163
\pgfpointtransformednonlinear, 1167
\pgfpointxy, 1088
\pgfpointxyz, 1089
\pgfpoptype, 1085
\pgfpositionnodelater, 1141
\pgfpositionnodelaterbox, 1141
\pgfpositionnodelatermaxx, 1141
\pgfpositionnodelatermaxy, 1141
\pgfpositionnodelaterminx, 1141
\pgfpositionnodelaterminy, 1141
\pgfpositionnodelatername, 1141
\pgfpositionnodelaterpath, 1142
\pgfpositionnodenow, 1142
\pgfprofileend, 762
\pgfprofileifisrunning, 762
\pgfprofilenew, 760
\pgfprofilenewforcommand, 761
\pgfprofilenewforcommandpattern, 761
\pgfprofilenewforenvironment, 762
\pgfprofilepostprocess, 762
\pgfprofilesetrel, 762
\pgfprofileshowinvocationsexpandedfor, 761
\pgfprofileshowinvocationsfor, 761
\pgfprofilestart, 762
\pgfpushtype, 1085
\pgfqbox, 1231
\pgfqboxsynced, 1231
\pgfqkeys, 979

\pgfqkeysactivatefamiliesandfilteroptions, 999
\pgfqkeysactivatesinglefamilyandfilteroptions, 999
\pgfqkeysalso, 979
\pgfqkeysfiltered, 997
\pgfqpoint, 1230
\pgfqpointscale, 1230
\pgfqpointxy, 1230
\pgfqpointxyz, 1230
\pgfrdfabout, 1087
\pgfrdfcontent, 1087
\pgfrdfdatatype, 1087
\pgfrdfhref, 1087
\pgfrdfinlist, 1087
\pgfrdfprefix, 1087
\pgfrdfproperty, 1087
\pgfrdfrel, 1087
\pgfrdfresource, 1087
\pgfrdfrev, 1087
\pgfrdfsrc, 1087
\pgfrdftypeof, 1087
\pgfrdfvocab, 1087
\pgfrealjobname, 1180
\pgfresetboundingbox, 1106
pgfscope environment, 1080
\pgfsetadditionalshadetransform, 1203
\pgfsetarrows, 1125
\pgfsetarrowsend, 1124
\pgfsetarrowsstart, 1124
\pgfsetbaseline, 1079
\pgfsetbaselinepointlater, 1079
\pgfsetbaselinepointnow, 1079
\pgfsetbeveljoin, 1122
\pgfsetblendmode, 1204
\pgfsetbuttcap, 1122
\pgfsetcolor, 1123
\pgfsetcornersarced, 1105
\pgfsetcurvilinearbeziercurve, 1170
\pgfsetdash, 1122
\pgfsetdecorationsegmenttransformation, 1117
\pgfseteorule, 1125
\pgfsetfading, 1206
\pgfsetfadingforcurrentpath, 1206
\pgfsetfadingforcurrentpathstroked, 1207
\pgfsetfillcolor, 1126
\pgfsetfillopacity, 1204
\pgfsetfillpattern, 1174
\pgfsetinnerlinewidth, 1124
\pgfsetinnerstrokecolor, 1124
\pgfsetlayers, 1193
\pgfsetlinetofirstplotpoint, 1190
\pgfsetlinewidth, 1122
\pgfsetmatrixcolumnsep, 1154
\pgfsetmatrixrowsep, 1155
\pgfsetmiterjoin, 1122
\pgfsetmiterlimit, 1122
\pgfsetmovetofirstplotpoint, 1190
\pgfsetnonzerorule, 1126
\pgfsetplotmarkphase, 756
\pgfsetplotmarkrepeat, 756
\pgfsetplotmarksize, 757
\pgfsetplottension, 749
\pgfsetrectcap, 1122
\pgfsetroundcap, 1122
\pgfsetroundjoin, 1122
\pgfsetshortenend, 1125
\pgfsetshortenstart, 1125
\pgfsetstrokecolor, 1123
\pgfsetstrokeopacity, 1204
\pgfsettransform, 1162
\pgfsettransformentries, 1163
\pgfsettransformnonlinearflatness, 1168
\pgfsetxvec, 1089
\pgfsetyvec, 1089
\pgfsetzvec, 1090
\pgfshadecolortocmyk, 1199
\pgfshadecolortogray, 1199
\pgfshadecolortorgb, 1198
\pgfshadepath, 1201
\pgfsnapshot, 1214
\pgfsnapshotafter, 1215
pgfsys package, 1234
pgfsys-common-pdf file, 1234
pgfsys-common-postscript file, 1234
pgfsys-dvi.def file, 122
pgfsys-dvipdfm.def file, 120
pgfsys-dvips.def file, 120
pgfsys-dvisvgm.def file, 120
pgfsys-pdftex.def file, 119
pgfsys-tex4ht.def file, 121
pgfsys-xetex.def file, 120
\pgfsys@animate, 1255
\pgfsys@animation@accesskey, 1270
\pgfsys@animation@accumulate, 1272
\pgfsys@animation@base, 1266
\pgfsys@animation@canvas@transform, 1261
\pgfsys@animation@event, 1269
\pgfsys@animation@freezeatend, 1271
\pgfsys@animation@movealong, 1264
\pgfsys@animation@noaccumulate, 1272
\pgfsys@animation@norotatealong, 1264
\pgfsys@animation@offset, 1269
\pgfsys@animation@removeatend, 1271
\pgfsys@animation@repeat, 1268
\pgfsys@animation@repeat@dur, 1268
\pgfsys@animation@repeat@event, 1270
\pgfsys@animation@repeat@indefinite, 1268
\pgfsys@animation@restart@always, 1271
\pgfsys@animation@restart@never, 1271
\pgfsys@animation@restart@whennotactive, 1271
\pgfsys@animation@rotatealong, 1264
\pgfsys@animation@syncbegin, 1269
\pgfsys@animation@syncend, 1269
\pgfsys@animation@time, 1265
\pgfsys@animation@tip@markers, 1259
\pgfsys@animation@val@color@cmy, 1267
\pgfsys@animation@val@color@cmyk, 1267
\pgfsys@animation@val@color@gray, 1267
\pgfsys@animation@val@color@rgb, 1267
\pgfsys@animation@val@current, 1266
\pgfsys@animation@val@dash, 1267
\pgfsys@animation@val@dimension, 1267
\pgfsys@animation@val@path, 1267
\pgfsys@animation@val@scalar, 1266
\pgfsys@animation@val@scale, 1267
\pgfsys@animation@val@text, 1266

\pgfsys@animation@val@translate, 1267
\pgfsys@animation@val@viewbox, 1267
\pgfsys@animation@whom, 1265
\pgfsys@append@type, 1246
\pgfsys@attach@to@id, 1246
\pgfsys@begin@idscope, 1246
\pgfsys@begininvisible, 1248
\pgfsys@beginpicture, 1235
\pgfsys@beginpurepicture, 1235
\pgfsys@beginscope, 1236
\pgfsys@beveljoin, 1240
\pgfsys@blend@mode, 1244
\pgfsys@buttcap, 1239
\pgfsys@clipfading, 1245
\pgfsys@clipnext, 1239
\pgfsys@closepath, 1237
\pgfsys@closestroke, 1238
\pgfsys@color@cmy, 1241
\pgfsys@color@cmy@fill, 1241
\pgfsys@color@cmy@stroke, 1241
\pgfsys@color@cmyk, 1240
\pgfsys@color@cmyk@fill, 1241
\pgfsys@color@cmyk@stroke, 1241
\pgfsys@color@gray, 1241
\pgfsys@color@gray@fill, 1241
\pgfsys@color@gray@stroke, 1241
\pgfsys@color@reset, 1241
\pgfsys@color@reset@inorderfalse, 1241
\pgfsys@color@reset@inordertrue, 1241
\pgfsys@color@rgb, 1240
\pgfsys@color@rgb@fill, 1240
\pgfsys@color@rgb@stroke, 1240
\pgfsys@color@unstacked, 1241
\pgfsys@curveto, 1236
\pgfsys@declarepattern, 1242
\pgfsys@defineimage, 1243
\pgfsys@definemask, 1245
\pgfsys@defobject, 1247
\pgfsys@discardpath, 1239
\pgfsys@end@idscope, 1246
\pgfsys@endinvisible, 1248
\pgfsys@endpicture, 1235
\pgfsys@endpurepicture, 1235
\pgfsys@endscope, 1236
\pgfsys@endviewbox, 1238
\pgfsys@fadingfrombox, 1244
\pgfsys@fill, 1238
\pgfsys@fill@opacity, 1244
\pgfsys@fillstroke, 1239
\pgfsys@functionalshading, 1243
\pgfsys@getposition, 1249
\pgfsys@global@papersize, 1248
\pgfsys@hbox, 1235
\pgfsys@hboxsynced, 1235
\pgfsys@horishading, 1243
\pgfsys@imagesuffixlist, 1242
\pgfsys@invoke, 1247
\pgfsys@lineto, 1236
\pgfsys@marker@declare, 1248
\pgfsys@marker@use, 1248
\pgfsys@markposition, 1249
\pgfsys@miterjoin, 1239
\pgfsys@moveto, 1236
\pgfsys@new@id, 1245
\pgfsys@opacity, 1244
\pgfsys@papersize, 1248
\pgfsys@patternmatrix, 1242
\pgfsys@pictureboxsynced, 1236
\pgfsys@pop@type, 1246
\pgfsys@push@type, 1246
\pgfsys@radialshading, 1243
\pgfsys@rdf@about, 1247
\pgfsys@rdf@content, 1247
\pgfsys@rdf@datatype, 1247
\pgfsys@rdf@href, 1247
\pgfsys@rdf@inlist, 1247
\pgfsys@rdf@prefix, 1247
\pgfsys@rdf@property, 1247
\pgfsys@rdf@rel, 1247
\pgfsys@rdf@resource, 1247
\pgfsys@rdf@rev, 1247
\pgfsys@rdf@src, 1247
\pgfsys@rdf@typeof, 1247
\pgfsys@rdf@vocab, 1247
\pgfsys@rect, 1237
\pgfsys@rectcap, 1239
\pgfsys@roundcap, 1239
\pgfsys@roundjoin, 1239
\pgfsys@setdash, 1240
\pgfsys@setlinewidth, 1239
\pgfsys@setmiterlimit, 1239
\pgfsys@setpatterncolored, 1242
\pgfsys@setpatternuncolored, 1242
\pgfsys@stroke, 1238
\pgfsys@stroke@opacity, 1244
\pgfsys@thepageheight, 1248
\pgfsys@thepagewidth, 1248
\pgfsys@transformcm, 1237
\pgfsys@transformshift, 1237
\pgfsys@transformxyscale, 1237
\pgfsys@transparencygroupfrombox, 1244
\pgfsys@typesetpicturebox, 1235
\pgfsys@use@id, 1246
\pgfsys@use@type, 1246
\pgfsys@usefading, 1244
\pgfsys@useobject, 1248
\pgfsys@vertshading, 1243
\pgfsys@viewboxmeet, 1237
\pgfsys@viewboxslice, 1238
\pgfsysanimate, 1255
\pgfsysanimkeyaccesskey, 1270
\pgfsysanimkeyaccumulate, 1272
\pgfsysanimkeybase, 1266
\pgfsysanimkeycanvastransform, 1261
\pgfsysanimkeyevent, 1269
\pgfsysanimkeyfreezeatend, 1271
\pgfsysanimkeymovealong, 1263
\pgfsysanimkeynoaccumulate, 1272
\pgfsysanimkeynorotatealong, 1264
\pgfsysanimkeyoffset, 1268
\pgfsysanimkeyremoveatend, 1271
\pgfsysanimkeyrepeat, 1268
\pgfsysanimkeyrepeatdur, 1268
\pgfsysanimkeyrepeatevent, 1270
\pgfsysanimkeyrepeatindefinite, 1268
\pgfsysanimkeyrestartalways, 1270

\pgfsysanimkeyrestartnever, 1271
\pgfsysanimkeyrestartwhennotactive, 1271
\pgfsysanimkeyrotatealong, 1264
\pgfsysanimkeysnapshotstart, 1257
\pgfsysanimkeysyncbegin, 1269
\pgfsysanimkeysyncend, 1269
\pgfsysanimkeytime, 1265
\pgfsysanimkeytipmarkers, 1259
\pgfsysanimkeywhom, 1265
\pgfsysanimsnapshot, 1256
\pgfsysanimsnapshotafter, 1257
\pgfsysanimvalcolorcmy, 1267
\pgfsysanimvalcolorcmyk, 1267
\pgfsysanimvalcolorgray, 1267
\pgfsysanimvalcolorrgb, 1267
\pgfsysanimvalcurrent, 1266
\pgfsysanimvaldash, 1267
\pgfsysanimvaldimension, 1267
\pgfsysanimvalpath, 1267
\pgfsysanimvalscalar, 1266
\pgfsysanimvalscale, 1267
\pgfsysanimvaltext, 1266
\pgfsysanimvaltranslate, 1267
\pgfsysanimvalviewbox, 1267
\pgfsysdriver, 1234
\pgfsysprotocol@bufferedfalse, 1254
\pgfsysprotocol@bufferedtrue, 1254
\pgfsysprotocol@flushcurrentprotocol, 1254
\pgfsysprotocol@getcurrentprotocol, 1254
\pgfsysprotocol@invokecurrentprotocol, 1254
\pgfsysprotocol@literal, 1254
\pgfsysprotocol@literalbuffered, 1254
\pgfsysprotocol@setcurrentprotocol, 1254
\pgfsyssoftpath@closepath, 1252
\pgfsyssoftpath@curveto, 1252
\pgfsyssoftpath@flushcurrentpath, 1252
\pgfsyssoftpath@getcurrentpath, 1252
\pgfsyssoftpath@invokecurrentpath, 1252
\pgfsyssoftpath@lineto, 1252
\pgfsyssoftpath@moveto, 1252
\pgfsyssoftpath@rect, 1252
\pgfsyssoftpath@setcurrentpath, 1252
\pgftext, 1082
\pgftransformarcaxesattime, 1161
\pgftransformarrow, 1160
\pgftransformationadjustments, 1163
\pgftransformcm, 1159
\pgftransformcurveattime, 1160
\pgftransforminvert, 1162
\pgftransformlineattime, 1160
\pgftransformnonlinear, 1166
\pgftransformreset, 1161
\pgftransformresetnontranslations, 1162
\pgftransformrotate, 1159
\pgftransformscale, 1158
\pgftransformshift, 1158
\pgftransformtriangle, 1159
\pgftransformxscale, 1158
\pgftransformxshift, 1158
\pgftransformxslant, 1158
\pgftransformyscale, 1158
\pgftransformyshift, 1158
\pgftransformyslant, 1159
pgftransparencygroup environment, 1207, 1208
\pgfuseid, 1085
\pgfuseimage, 1176
\pgfusepath, 1121
\pgfusepathqclip, 1231
\pgfusepathqfill, 1231
\pgfusepathqfillstroke, 1231
\pgfusepathqstroke, 1231
\pgfuseplotmark, 756
\pgfuseshading, 1201
\pgfusetype, 1085
\pgfutil@tempboxa, 1229
\pgfutil@tempdima, 1228
\pgfutil@tempdimb, 1228
\pgfverticaltransformationadjustment, 1163
pgfviewboxscope environment, 1165, 1166
\pgfwarning, 1087
phase key, 306, 891
phylogenetic edge key, 483
phylogenetic inner node key, 483
phylogenetic tree by author key, 484
phylogenetic tree layout key, 483
phylogenetics graph drawing library, 483
pi math function, 1039
\pic, 261
.pic handler, 266
pic path operation, 261
pic actions key, 263
pic text key, 265
pic text options key, 265
pic type key, 262
Pic Types
    angle, 568
    cube folding, 725
    cube truncated folding, 725
    cuboctahedron folding, 727
    cuboctahedron truncated folding, 727
    dodecahedron folding, 726
    dodecahedron golden folding, 727
    dodecahedron rhombic folding, 727
    dodecahedron' folding, 726
    icosahedron folding, 728
    icosidodecahedron folding, 729
    octahedron folding, 725, 726
    rhombicuboctahedron folding, 728
    right angle, 569
    snub cube folding, 728
    tetrahedron folding, 724
    tetrahedron truncated folding, 725
pin key, 251
pin distance key, 251
pin edge key, 251
pin position key, 251
place key, 744
placement/chain count key, 310
placement/compute position key, 310
placement/depth key, 310
placement/element count key, 308
placement/level key, 307
placement/logical node depth key, 310
placement/logical node width key, 309
placement/place key, 311
placement/width key, 308

plane origin key, 565
plane x key, 565
plane y key, 565
plot module, 1186
plot path operation, 341
Plot marks
*, 757
|, 758
+, 757
-, 758
10-pointed star, 758
asterisk, 758
ball, 347
diamond, 758
diamond*, 758
halfcircle, 758
halfcircle*, 758
halfdiamond*, 758
halfsquare left*, 758
halfsquare right*, 758
halfsquare*, 758
heart, 758
Mercedes star, 758
Mercedes star flipped, 758
o, 758
oplus, 758
oplus*, 758
otimes, 758
otimes*, 758
pentagon, 758
pentagon*, 758
square, 758
square*, 758
star, 758
text, 758
triangle, 758
triangle*, 758
x, 757
plothandlers library, 749
plotmarks library, 758
point key, 141
point down key, 613
point left key, 613
point right key, 613
point up key, 612
pointCloud (Lua), 508
points key, 735
polar comb key, 351
polyline layer edge routing key, 472
pos key, 244
position animation attribute, 403
positioning library, 238
post key, 371, 745
post length key, 372
postaction key, 188
pow math function, 1036
pre key, 371, 745
pre and post key, 745
pre length key, 371
preaction key, 186
precision key, 1058
Predefined node
⟨chain name⟩-begin, 603
⟨chain name⟩-end, 603
current bounding box, 1144
current page, 1145
current path bounding box, 1144
current subpath start, 1144
data bounding box, 862
data visualization bounding box, 862
path picture bounding box, 181
.prefix handler, 990
prefix key, 346, 672, 763
prefix after command key, 153
.prefix code handler, 988
.prefix style handler, 989
prepareBoundingBoxes (Lua), 522
prepareRotateAround (Lua), 522
print tikz math function, 708
print sign key, 1061
prioritized greedy cycle removal key, 471
PriorityQueue (Lua), 538
PriorityQueue.
new (Lua), 538
PriorityQueue:
dequeue (Lua), 538
enqueue (Lua), 538
isEmpty (Lua), 538
updatePriority (Lua), 538
probability key, 534
profiler library, 760
pushLayout (Lua), 554
pushOption (Lua), 554
pushSubgraphVertex (Lua), 553
put node text on incoming edges key, 291
put node text on outgoing edges key, 291

q key, 740
quarter about strategy key, 893
quick key, 201, 285
quotes library, 252
quotes mean label key, 253
quotes mean pin key, 254

r key, 740
r postfix math operator, 1033
\r@pgf@reada, 1229
rad math function, 1040
radial shading, 778
radius key, 137, 157, 306, 449, 564, 637, 735
raise key, 370
rand math function, 1043
random (Lua), 537
random math function, 1043
random greedy cycle removal key, 471
random seed key, 448
random starburst key, 804
random steps decoration, 639
random_permutation (Lua), 535
randomize angle percent key, 699
randomize step percent key, 699
randomseed (Lua), 537
range key, 345
raw gnuplot key, 346
Rays arrow tip, 221
rdf library, 763
rdf engine key, 763

rdf engine on key, 763
read comma as period key, 1058
read from file key, 859, 869
real math function, 1039
real tikz math function, 705
recolor ⟨color class name⟩ by key, 292
reconnect (Lua), 502
Rectangle arrow tip, 218
rectangle path operation, 156
rectangle shape, 786
rectangle callout shape, 825
rectangle ee shape, 631
rectangle split shape, 820
rectangle split allocate boxes key, 821
rectangle split draw splits key, 822
rectangle split empty part depth key, 821
rectangle split empty part height key, 821
rectangle split empty part width key, 821
rectangle split horizontal key, 821
rectangle split ignore empty parts key, 821
rectangle split part align key, 821
rectangle split part fill key, 822
rectangle split parts key, 821
rectangle split use custom fill key, 822
rectangular phylogram key, 486
regardless at key, 432
regular polygon shape, 791
regular polygon sides key, 791
rel thresh key, 696
relationship key, 667
relative key, 840
relative style key, 1056
relative* key, 1054
remake next key, 674
remember key, 387, 1007
remember picture key, 258
remove (Lua), 501, 537
render_collections (Lua), 555
render_edges (Lua), 556
render_vertices (Lua), 555
renderCollection (Lua), 558
renderCollectionStartKind (Lua), 558
renderCollectionStopKind (Lua), 558
renderEdge (Lua), 559
renderEdgesStart (Lua), 559
renderEdgesStop (Lua), 559
renderGraph (Lua), 555
renderStart (Lua), 558
renderStop (Lua), 558
renderVertex (Lua), 558
renderVerticesStart (Lua), 558
renderVerticesStop (Lua), 558
repeat key, 1227
repeat state key, 1112
repeats key, 1226
reset cm key, 378
reset marks key, 648
resistor key, 626, 629
resize to layout, 1017
restart key, 1226
resume key, 387
resumeGraphDrawingCoroutine (Lua), 552
retain unit mantissa key, 1062
.retry handler, 994
return tikz math function, 707
reverse path key, 656
reversed (Lua), 516
reversed key, 196
rhombicuboctahedron folding pic type, 728
right key, 197, 238, 241, 849, 1082
right anchor key, 277
right angle key, 699
right angle pic type, 569
right color key, 776
right delimiter key, 712
rnd math function, 1043
root key, 466
root concept key, 716
rooted rectangular phylogram key, 485
rooted straight phylogram key, 486
rotate animation attribute, 399
rotate basic layer animation attribute, 1221
rotate key, 377, 1084
rotate system layer animation attribute, 1263
rotate along key, 1223
rotate around key, 377
rotate around x key, 377
rotate around y key, 378
rotate around z key, 378
rotate fit key, 685
round key, 200
round math function, 1038
Round Cap arrow tip, 220
rounded corners key, 156
rounded corners layout, 1018
rounded rectangle shape, 829
rounded rectangle arc length key, 829
rounded rectangle east arc key, 829
rounded rectangle left arc key, 829
rounded rectangle right arc key, 829
rounded rectangle west arc key, 829
routing graph drawing library, 488
row ⟨number⟩ key, 323
row ⟨row number⟩ column ⟨column number⟩ key, 323
row sep key, 321
rt key, 849
\rule, 700
rule set key, 702
runGraphDrawingAlgorithm (Lua), 552

same layer key, 469
samples key, 343, 867
samples at key, 343
Saturday date test, 1010
save path key, 170
\savedanchor, 1147
\saveddimen, 1147
\savedmacro, 1148
saw decoration, 639
scalar math function, 1032
scale (Lua), 514
scale animation attribute, 399
scale basic layer animation attribute, 1221
scale key, 195, 376
scale system layer animation attribute, 1263
scale around key, 376
scale file plot x key, 688, 690

scale file plot y key, 688, 690
scale file plot z key, 688, 690
scale length key, 195
scale results key, 687, 690
scale width key, 195
school book axes key, 885
school book axes/standard labels key, 886
school book axes/unit key, 886
Schottky diode key, 629
sci key, 1053
sci 10^e key, 1061
sci 10e key, 1061
sci E key, 1061
sci e key, 1061
sci generic key, 1061
sci generic/exponent key, 1062
sci generic/mantissa sep key, 1062
sci precision key, 1058
sci subscript key, 1061
sci superscript key, 1061
sci zerofill key, 1054
scientific axes key, 882
scientific axes/clean key, 884
scientific axes/end labels key, 885
scientific axes/height key, 883
scientific axes/inner ticks key, 884
scientific axes/outer ticks key, 883
scientific axes/standard labels key, 884
scientific axes/upright labels key, 885
scientific axes/width key, 883
scientific polar axes key, 965
scientific polar axes/-180 to 180 key, 968
scientific polar axes/-90 to 0 key, 968
scientific polar axes/-90 to 90 key, 968
scientific polar axes/-pi half to 0 key, 969
scientific polar axes/-pi half to pi half key, 969
scientific polar axes/-pi to pi key, 969
scientific polar axes/0 to 180 key, 968
scientific polar axes/0 to 2pi key, 969
scientific polar axes/0 to 360 key, 968
scientific polar axes/0 to 90 key, 968
scientific polar axes/0 to pi key, 969
scientific polar axes/0 to pi half key, 969
scientific polar axes/clean key, 967
scientific polar axes/fourth quadrant key, 970
scientific polar axes/fourth quadrant clockwise key, 970
scientific polar axes/inner ticks key, 966
scientific polar axes/left half key, 971
scientific polar axes/left half clockwise key, 971
scientific polar axes/lower half key, 970
scientific polar axes/lower half clockwise key, 970
scientific polar axes/outer ticks key, 966
scientific polar axes/quadrant key, 970
scientific polar axes/quadrant clockwise key, 970
scientific polar axes/right half key, 971
scientific polar axes/right half clockwise key, 971
scientific polar axes/upper half key, 970
scientific polar axes/upper half clockwise key, 970
Scope (Lua), 497
scope (data visualization), 861
scope environment, 131, 132
scope key, 388
scope fading key, 362
scope is new context key, 767
Scope.
    new (Lua), 498
\scoped, 132
scopes library, 132
.search also handler, 992
sec math function, 1040
second key, 465
segment length key, 637
semicircle shape, 790
semithick key, 173
semitransparent key, 354
sep key, 206
separator key, 866
set key, 282, 922, 923
set decimal separator key, 1058
set thousands separator key, 1059
set up code key, 732
setPolylinePath (Lua), 510, 511
\shade, 171
shade key, 182
\shadedraw, 171
shades of blue style sheet, 940
shades of red style sheet, 940
shading key, 182
shading angle key, 182
Shadings
    axis, 776
    ball, 776
    bilinear interpolation, 777
    color wheel, 777
    color wheel black center, 778
    color wheel white center, 778
    Mandelbrot set, 778
    radial, 778
shadings library, 776
shadow scale key, 780
shadow xshift key, 780
shadow yshift key, 781
shadows library, 780
shape key, 224, 652
shape aspect key, 229
shape backgrounds decoration, 651
shape border rotate key, 230
shape border uses incircle key, 230
shape end height key, 654
shape end size key, 654
shape end width key, 654
shape evenly spread key, 653
shape height key, 651
shape scaled key, 654
shape sep key, 653
shape size key, 651
shape sloped key, 653
shape start height key, 654
shape start size key, 654

shape start width key, 654
shape width key, 651
Shapes
arrow box, 813
battery IEC, 635
break contact IEC, 636
breakdown diode IEC, 633
capacitor IEC, 635
chamfered rectangle, 830
circle, 785
circle ee, 631
circle solidus, 817
circle split, 816
circular sector, 798
cloud, 802
cloud callout, 826
correct forbidden sign, 801
cross out, 828
cylinder, 800
dart, 797
diamond, 786
diamond split, 819
direction ee, 631
double arrow, 812
ellipse, 787
ellipse callout, 825
ellipse split, 818
forbidden sign, 802
generic circle IEC, 632
generic diode IEC, 632
ground IEC, 636
inductor IEC, 634
isosceles triangle, 794
kite, 795
magnetic tape, 808
magnifying glass, 802
make contact IEC, 636
nand gate IEC, 625
nand gate US, 623
rectangle, 786
rectangle callout, 825
rectangle ee, 631
rectangle split, 820
regular polygon, 791
rounded rectangle, 829
semicircle, 790
signal, 805
single arrow, 810
star, 792
starburst, 804
strike out, 828
tape, 807
trapezium, 788
var make contact IEC, 636
var resistor IEC, 634
shapes module, 1137
shapes.arrows library, 810
shapes.callouts library, 823
shapes.gates.ee library, 630
shapes.gates.ee.IEC library, 632
shapes.gates.logic library, 620
shapes.gates.logic.IEC library, 623
shapes.gates.logic.US library, 622
shapes.geometric library, 786
shapes.misc library, 827
shapes.multipart library, 816
shapes.symbols library, 801
sharp key, 200
sharp corners key, 157
sharp plot key, 348
shell escape key, 671
shift (Lua), 513, 516
shift animation attribute, 400
shift key, 375
shift only key, 375
shiftByCoordinate (Lua), 514, 516
shorten < key, 209, 1220
shorten < key, 398
shorten > key, 209, 1220
shorten > key, 398
show background bottom key, 580
show background grid key, 578
show background left key, 580
show background rectangle key, 577
show background right key, 580
show background top key, 579
.show code handler, 995
show path construction decoration, 643
.show value handler, 994
showpos key, 1061
sibling angle key, 846
sibling distance key, 336, 424
sibling post sep key, 424
sibling pre sep key, 424
sibling sep key, 424
siemens key, 630
sign math function, 1037
signal class, 1070
signal method, 1071
signal shape, 805
signal from key, 806
signal pointer angle key, 806
signal to key, 806
significant sep key, 461
silent key, 1026
simple key, 286
simple demo layout key, 530
simple edge demo layout key, 531
simple Huffman layout key, 532
simple necklace layout key, 481
sin math function, 1040
sin path operation, 162
single arrow shape, 810
single arrow head extend key, 810
single arrow head indent key, 811
single arrow tip angle key, 810
sinh math function, 1045
sixpointed stars pattern, 730
size key, 833
skew x basic layer animation attribute, 1222
skew y basic layer animation attribute, 1222
skewx system layer animation attribute, 1263
skewy system layer animation attribute, 1263
skip 0. key, 1060
slant key, 196
slice key, 851

sloped key, 246
small circuit symbols key, 610
small components first key, 435
small mindmap key, 714
smooth key, 348
smooth cycle key, 349
snake decoration, 641
snub cube folding pic type, 728
solid key, 175
solution key, 142
some key, 893
sort by key, 145
sortIncoming (Lua), 502
sortOutgoing (Lua), 502
source edge clear key, 290
source edge node key, 290
source edge style key, 290
south fading, 683
span edge key, 466
span priority key, 466
span priority -> key, 467
span priority reversed -> key, 467
span using all key, 467
span using directed key, 467
spanPriority (Lua), 509
spring constant key, 476
spring electrical Hu 2006 layout key, 479
spring electrical layout key, 479
spring electrical layout' key, 479
spring electrical Walshaw 2000 layout key, 480
spring Hu 2006 layout key, 479
spring layout key, 479
\spy, 833
spy library, 832
spy connection path key, 835
spy scope key, 833
spy using outlines key, 836
spy using overlays key, 836
sqrt math function, 1036
Square arrow tip, 218
square plot mark, 758
square* plot mark, 758
stack key, 904
stack' key, 904
stage animation attribute, 395
stage basic layer animation attribute, 1218
standard about strategy key, 892
star plot mark, 758
star shape, 792
star point height key, 793
star point ratio key, 793
star points key, 793
starburst shape, 804
starburst point height key, 804
starburst points key, 804
Stars pattern, 735
start angle key, 159
start branch key, 605
start chain key, 600
start radius key, 637
\state, 1110, 1118
state key, 572
state with output key, 572
state without output key, 572
statement key, 764
statements/has as member key, 768
statements/has type key, 766
statements/is a bag key, 768
statements/is a container key, 768
statements/is a sequence key, 768
statements/is an alternative key, 769
statements/object key, 765
statements/predicate key, 765
statements/subject key, 765
status key, 1026
stay key, 1213
std key, 1054
Stealth arrow tip, 218
step key, 160, 699, 890, 1103
stepx key, 1103
stepy key, 1103
Storage (Lua), 519
Storage.
    new (Lua), 519
    newTableStorage (Lua), 519
store key, 863
.store in handler, 990
store in set key, 871
Straight Barb arrow tip, 213
straight phylogram key, 486
straight zigzag decoration, 638
stride length key, 650
strike out shape, 828
stroke opacity animation attribute, 395
strokecolor system layer animation attribute, 1258
strokeopacity system layer animation attribute, 1258
strong colors style sheet, 939
structured tokens key, 747
.style handler, 988
style key, 896
.style 2 args handler, 988
.style args handler, 989
.style n args handler, 988
.style sheet handler, 933
style sheet key, 933
Style sheets
    cross marks, 938
    gray scale, 940
    shades of blue, 940
    shades of red, 940
    strong colors, 939
    vary dashing, 937
    vary hue, 940
    vary thickness, 937
    vary thickness and dashing, 938
style sheets/⟨*style sheet*⟩/default style key, 934
Styles for graphics, *see* Graphic options and styles
styling key, 896
subgraph C_n graph, 313
subgraph Grid_n graph, 313
subgraph I_n graph, 311
subgraph I_nm graph, 312
subgraph K_n graph, 312
subgraph K_nm graph, 313
subgraph nodes key, 445

subgraph P_n graph, 313
subgraph text bottom key, 447
subgraph text none key, 446
subgraph text sep key, 447
subgraph text top key, 446
subminor key, 894
subminor also at key, 896
subminor at key, 895
subtract math function, 1035
summary (Lua), 494
Sunday date test, 1010
svg path operation, 163
svg.path library, 838
swap key, 197, 245
sweep crossing minimization key, 472
switch if input segment less than key, 1111
switch if less than key, 1111, 1118
\symbol, 698
sync (Lua), 503, 509
sync key, 389
syntacticTailAndHead (Lua), 508
system call key, 671

table format, 865
tail anchor key, 442
tail cut key, 442
tailAnchorForArcPath (Lua), 509
tailAnchorForEdgePath (Lua), 511
tan math function, 1040
tangent coordinate system, 141
tanh math function, 1046
tape shape, 807
tape bend bottom key, 807
tape bend height key, 807
tape bend top key, 807
target edge clear key, 289
target edge node key, 289
target edge style key, 289
Tee Barb arrow tip, 214
tension key, 348
tetrahedron folding pic type, 724
tetrahedron truncated folding pic type, 725
TeX code format, 867
text animation attribute, 395
text key, 231, 655, 657
text plot mark, 758
text align key, 656, 658
text align/align key, 656
text align/center key, 656
text align/fit to path key, 657
text align/fit to path stretching spaces key, 657
text align/left key, 656
text align/left indent key, 656
text align/right key, 656
text align/right indent key, 656
text along path decoration, 655
text color key, 656
text depth key, 236
text effects key, 658
text effects along path decoration, 657
text effects/character ⟨*number*⟩ key, 659
text effects/character command key, 664
text effects/character count key, 660
text effects/character total key, 660
text effects/character widths key, 659
text effects/characters key, 659
text effects/every character key, 658
text effects/every character width key, 659
text effects/every first letter key, 659
text effects/every last letter key, 659
text effects/every letter key, 659
text effects/every word key, 659
text effects/every word separator key, 659
text effects/fit text to path key, 662
text effects/group letters key, 663
text effects/letter ⟨*number*⟩ key, 659
text effects/letter count key, 660
text effects/path from text key, 661
text effects/path from text angle key, 662
text effects/repeat text key, 664
text effects/replace characters key, 664
text effects/reverse text key, 663
text effects/scale text to path key, 662
text effects/style characters key, 661
text effects/text along path key, 658
text effects/word ⟨*m*⟩ letter ⟨*n*⟩ key, 659
text effects/word ⟨*number*⟩ key, 659
text effects/word count key, 661
text effects/word separator key, 659
text effects/word total key, 661
text format delimiters key, 656
text height key, 236
text mark key, 758
text mark as node key, 759
text mark style key, 759
text opacity key, 355
text width key, 233
text/effetcs/letter total key, 661
thick key, 173
thin key, 173
third key, 465
three point perspective coordinate system, 738
through library, 844
Thursday date test, 1010
tick layer key, 899
tick length key, 914
tick node layer key, 899
tick prefix key, 901
tick suffix key, 901
tick text at high key, 915
tick text at low key, 915
tick text even padding key, 904
tick text high even padding key, 904
tick text high odd padding key, 904
tick text low even padding key, 904
tick text low odd padding key, 904
tick text odd padding key, 904
tick text padding key, 904
tick typesetter key, 902
tick unit key, 901
ticks decoration, 642
tight background key, 578
\tikz, 130
/tikz/
    ', 246
    >, 209

3d view, 738
above, 237, 238
above delimiter, 712
above left, 238, 241
above right, 238, 242
absolute, 249
accepting, 573
accepting above, 574
accepting below, 574
accepting by arrow, 573
accepting by double, 573
accepting left, 574
accepting right, 574
accepting text, 573
accepting where, 574
alias, 224
align, 233
allow upside down, 247
ampersand replacement, 326
anchor, 139, 237, 326
and gate, 619
angle eccentricity, 568
angle radius, 568
animate/
  arrows, 398
  attribute, 385
  base, 404
  entry, 383
  fork, 387
  id, 385
  object, 384
  options/along, 400
  options/begin, 405
  options/begin on, 405
  options/begin snapshot, 413
  options/ease, 411
  options/ease in, 411
  options/ease out, 411
  options/end, 405
  options/end on, 409
  options/entry control, 411
  options/exit control, 410
  options/forever, 405
  options/freeze, 405
  options/jump, 412
  options/origin, 401
  options/repeat, 410
  options/repeats, 409
  options/restart, 408
  options/stay, 412
  options/transform, 402
  remember, 387
  resume, 387
  scope, 388
  shorten < , 398
  shorten > , 398
  sync, 389
  time, 386
  value, 388
animate, 382
animations/
  base, 1213
  begin snapshot, 1215
annotation, 721
annotation arrow, 615, 628
append after command, 153
arrows, 189, 209
at, 157, 223
at end, 247
at start, 247
attribute, 668
auto, 245
background grid, 579
background rectangle, 578
background top, 580
badness warnings for centered text, 235
ball color, 777
base left, 242
base right, 242
baseline, 129
behind path, 223
below, 238, 240
below delimiter, 712
below left, 238, 242
below right, 238, 242
bend, 161
bend angle, 841
bend at end, 162
bend at start, 162
bend left, 840
bend pos, 161
bend right, 841
blend group, 356
blend mode, 355
bottom color, 776
callout absolute pointer, 824
callout relative pointer, 824
canvas is plane, 565
canvas is xy plane at z, 565
canvas is xz plane at y, 565
canvas is yx plane at z, 565
canvas is yz plane at x, 565
canvas is zx plane at y, 565
canvas is zy plane at x, 566
cells, 322
centered, 238
chain default direction, 600
child anchor, 340
children are tokens, 745
circle connection bar, 719
circle connection bar switch color, 720
circle through, 844
circuit declare annotation, 615
circuit declare symbol, 610
circuit declare unit, 614
circuit ee, 626
circuit ee IEC, 626
circuit handle symbol, 611
circuit logic, 618
circuit logic CDH, 619
circuit logic IEC, 618
circuit logic US, 618
circuit symbol filled, 616
circuit symbol lines, 616
circuit symbol open, 616
circuit symbol size, 610

circuit symbol unit, 610
circuit symbol wires, 616
circuits, 610
circular drop shadow, 782
circular glow, 783
clip, 185
clockwise from, 846
cm, 378
color, 172
colored tokens, 746
column ⟨*number*⟩, 323
column sep, 320
concept, 714
concept color, 715, 716
concept connection, 717
connect spies, 836
const plot, 349
const plot mark left, 349
const plot mark mid, 350
const plot mark right, 349
continue branch, 605
continue chain, 601
controls, 842
copy shadow, 781
counterclockwise from, 846
cs/
  angle, 137, 140, 564
  horizontal line through, 143
  latitude, 564
  longitude, 564
  name, 139
  node, 141
  point, 141
  radius, 137, 564
  solution, 142
  vertical line through, 143
  x, 136, 738
  x radius, 137, 138
  y, 136, 738
  y radius, 137, 138
  z, 136, 564, 738
current point is local, 147
curve to, 839
dash, 175
dash dot, 176
dash dot dot, 176
dash expand off, 175
dash pattern, 174
dash phase, 175
dashed, 175
data visualization/
  ⟨*axis name*⟩, 874
  ⟨*axis system name*⟩, 919
  ⟨*legend name*⟩, 945
  ⟨*visualizer name*⟩, 923
  about, 891
  about strategy, 891
  after creation, 864
  after survey, 863
  after visualization, 863
  all axes, 874
  also at, 895
  arg1, 864
arg1 from key, 864
arg1 handle from key, 864
at, 895
at end survey, 863
at end visualization, 863
at start survey, 863
at start visualization, 863
axis layer, 910
axis option/anchor at max, 917
axis option/anchor at min, 917
axis options/attribute, 874
axis options/degrees, 972
axis options/exponential steps, 905
axis options/function, 877
axis options/goto, 908
axis options/goto pos, 909
axis options/grid, 889
axis options/include value, 875
axis options/label, 880
axis options/length, 879
axis options/linear steps, 905
axis options/logarithmic, 878
axis options/max value, 876
axis options/min value, 875
axis options/padding, 911
axis options/padding max, 911
axis options/padding min, 910
axis options/power unit length, 880
axis options/radians, 972
axis options/scaling, 876
axis options/scaling/default, 878
axis options/tick placement strategy, 906
axis options/ticks, 888
axis options/ticks and grid, 889
axis options/unit length, 879
axis options/unit vector, 881
axis options/unit vectors, 972
axis options/visualize axis, 908
axis options/visualize grid, 911
axis options/visualize label, 916
axis options/visualize ticks, 913
before creation, 863
before survey, 863
before visualization, 863
class, 863
common, 894
compute step, 906
data point, 860
decimal about strategy, 893
direction axis, 912
euro about strategy, 892
every ⟨*axis system name*⟩, 919
every axis, 910
every data set label, 943
every grid, 897
every label in data, 943
every label in legend, 956
every major grid, 898
every major ticks, 899
every minor grid, 898
every minor ticks, 899
every scientific axes, 883
every subminor grid, 898

every subminor ticks, 899
every ticks, 899
every visualizer, 925
few, 893
grid layer, 897
half about strategy, 892
high, 910, 914
int about strategy, 893
legend, 946
legend entry options/circular label in legend line, 962
legend entry options/default label in legend closed path, 959
legend entry options/default label in legend mark, 959
legend entry options/default label in legend path, 959
legend entry options/gap circular label in legend line, 962
legend entry options/label in legend line coordinates, 960
legend entry options/label in legend mark coordinates, 961
legend entry options/label in legend one mark, 962
legend entry options/label in legend three marks, 963
legend entry options/legend, 946
legend entry options/node style, 956
legend entry options/setup, 958
legend entry options/straight label in legend line, 961
legend entry options/text, 947
legend entry options/text colored, 956
legend entry options/text left, 957
legend entry options/text only, 957
legend entry options/text right, 957
legend entry options/visualizer in legend, 958
legend entry options/visualizer in legend style, 958
legend entry options/zig zag label in legend line, 961
legend options/above, 953
legend options/above left of, 955
legend options/above of, 955
legend options/above right of, 955
legend options/anchor, 951
legend options/at, 951
legend options/at values, 955
legend options/below, 953
legend options/below left of, 955
legend options/below of, 955
legend options/below right of, 955
legend options/columns, 950
legend options/down then left, 948
legend options/down then right, 947
legend options/east inside, 954
legend options/east outside, 951
legend options/every legend inside, 954
legend options/every new legend, 945
legend options/ideal number of columns, 950
legend options/ideal number of rows, 950
legend options/label style, 956
legend options/left, 952
legend options/left of, 955
legend options/left then down, 948
legend options/left then up, 948
legend options/matrix node style, 945
legend options/max columns, 949
legend options/max rows, 949
legend options/north east inside, 954
legend options/north east outside, 952
legend options/north inside, 954
legend options/north outside, 953
legend options/north west inside, 954
legend options/north west outside, 952
legend options/opaque, 954
legend options/right, 951
legend options/right of, 955
legend options/right then down, 948
legend options/right then up, 948
legend options/rows, 950
legend options/south east inside, 953
legend options/south east outside, 952
legend options/south inside, 954
legend options/south outside, 953
legend options/south west inside, 954
legend options/south west outside, 953
legend options/transparent, 954
legend options/up then left, 948
legend options/up then right, 948
legend options/west inside, 954
legend options/west outside, 952
low, 909, 914
major, 894
major also at, 895
major at, 895
many, 893
minor, 894
minor also at, 895
minor at, 895
minor steps between steps, 890
new axis base, 873
new axis system, 919
new Cartesian axis, 881
new legend, 945
new legend entry, 958
new object, 863
new polar axes, 971
new visualizer, 928
no tick text, 915
no tick text at, 900
node style, 896
node styling, 897
none, 894
options at, 900
padded, 910
phase, 891
quarter about strategy, 893
school book axes, 885
school book axes/standard labels, 886
school book axes/unit, 886
scientific axes, 882
scientific axes/clean, 884

scientific axes/end labels, 885
scientific axes/height, 883
scientific axes/inner ticks, 884
scientific axes/outer ticks, 883
scientific axes/standard labels, 884
scientific axes/upright labels, 885
scientific axes/width, 883
scientific polar axes, 965
scientific polar axes/clean, 967
scientific polar axes/inner ticks, 966
scientific polar axes/outer ticks, 966
some, 893
stack, 904
stack', 904
standard about strategy, 892
step, 890
store, 863
style, 896
style sheet, 933
styling, 896
subminor, 894
subminor also at, 896
subminor at, 895
tick layer, 899
tick length, 914
tick node layer, 899
tick prefix, 901
tick suffix, 901
tick text at high, 915
tick text at low, 915
tick text even padding, 904
tick text high even padding, 904
tick text high odd padding, 904
tick text low even padding, 904
tick text low odd padding, 904
tick text odd padding, 904
tick text padding, 904
tick typesetter, 902
tick unit, 901
uv axes, 887
uv Cartesian, 887
uvw axes, 887
uvw Cartesian cabinet, 887
visualizer label options/auto, 942
visualizer label options/index, 942
visualizer label options/node style, 943
visualizer label options/pin angle, 944
visualizer label options/pin length, 944
visualizer label options/pos, 942
visualizer label options/text, 941
visualizer label options/text colored, 943
visualizer label options/text', 941
visualizer label options/when, 941
visualizer options/gap cycle, 927
visualizer options/gap line, 927
visualizer options/ignore style sheets, 924
visualizer options/label in data, 941
visualizer options/label in legend, 946
visualizer options/label in legend options, 959
visualizer options/no lines, 927
visualizer options/pin in data, 943
visualizer options/polygon, 926
visualizer options/smooth cycle, 927
visualizer options/smooth line, 926
visualizer options/straight cycle, 926
visualizer options/straight line, 926
visualizer options/style, 924
when, 863
xy axes, 887
xy Cartesian, 887
xyz axes, 887
xyz Cartesian cabinet, 887
data visualizers/
visualize as line, 925
visualize as scatter, 928
visualize as smooth line, 926
dates, 582
day code, 584
day list downward, 590
day list left, 591
day list right, 591
day list upward, 591
day text, 585
day xshift, 583
day yshift, 583
decorate, 369
delta angle, 159
densely dash dot, 176
densely dash dot dot, 176
densely dashed, 176
densely dotted, 175
distance, 842
domain, 343
dotted, 175
double, 177
double copy shadow, 782
double distance, 177
double distance between line centers, 177
double equal sign distance, 177
draw, 172
draw opacity, 353
drop shadow, 781
edge from parent, 339
edge from parent fork down, 847
edge from parent fork left, 847
edge from parent fork right, 847
edge from parent fork up, 847
edge from parent macro, 340
edge from parent path, 339
edge label, 164
edge label', 164
edge node, 164
end angle, 159
entity, 667
evaluate, 703
even odd rule, 180
every ⟨*part name*⟩ node part, 231
every ⟨*shape*⟩ node, 225
every above delimiter, 712
every accepting by arrow, 574
every annotation, 722
every attribute, 668
every below delimiter, 712

every calendar, 582
every cell, 322
every child, 334
every child node, 334
every circle, 158
every circle connection bar, 719
every circuit ee, 626
every circuit logic, 618
every circuit symbol, 611
every concept, 715
every cut, 724
every data, 860
every day, 585
every delimiter, 712
every edge, 257
every edge quotes, 257
every entity, 667
every even column, 323
every even row, 323
every extra concept, 715
every fit, 684
every fold, 724
every info, 613
every initial by arrow, 573
every join, 604
every label, 250
every label quotes, 254
every left delimiter, 712
every loop, 843
every mark, 347
every matrix, 318
every mindmap, 713
every month, 586
every new --, 277
every new ->, 276
every new <-, 277
every new <->, 277
every node, 225
every odd column, 323
every odd row, 323
every on background layer, 577
every on chain, 603
every outer matrix, 318
every path, 153
every pic, 264
every pic quotes, 266
every picture, 130
every pin, 251
every pin edge, 251
every pin quotes, 254
every place, 744
every plot, 346
every relationship, 668
every right delimiter, 712
every scope, 131
every shadow, 781
every spy in node, 834
every spy on node, 835
every state, 572
every subgraph node, 445
every to, 165, 166
every token, 745
every transition, 744
every year, 587
execute after day scope, 589
execute at begin cell, 325
execute at begin day scope, 588
execute at begin node, 226
execute at begin picture, 129
execute at begin scope, 131
execute at begin to, 166
execute at empty cell, 325
execute at end cell, 325
execute at end day scope, 589
execute at end node, 226
execute at end picture, 129
execute at end scope, 131
execute at end to, 166
execute before day scope, 588
external/
  aux in dpth, 671
  disable dependency files, 674
  export, 674
  export next, 674
  figure list, 676
  figure name, 673
  force remake, 674
  mode, 676
  only named, 678
  optimize, 677
  optimize command away, 677
  optimize/install, 678
  optimize/restore, 678
  prefix, 672
  remake next, 674
  shell escape, 671
  system call, 671
  up to date check, 675
  verbose, 677
  verbose IO, 677
  verbose optimize, 677
extra concept, 715
face 1, 724
face 2, 724
face 3, 724
face 4, 724
fading angle, 361
fading transform, 360
fill, 178
fill opacity, 354
fit, 684
fit fading, 360
folding line length, 724
font, 232
framed, 578
general shadow, 780
graph/
  level, 307
  level ⟨*level*⟩, 307
  parse, 275
graphs/
  --, 294
  ->, 294
  -!-, 294
  <-, 294
  <->, 294

⟨*color class name*⟩, 292
as, 280
branch down, 302
branch down sep, 305
branch left, 302
branch left sep, 304
branch right, 302
branch right sep, 304
branch up, 301
branch up sep, 305
butterfly, 316
butterfly/from, 317
butterfly/level, 317
butterfly/to, 317
Cartesian placement, 300
chain polar shift, 305
chain shift, 300
circular placement, 305
clear <, 290
clear >, 290
clique, 314
clockwise, 307
color class, 292
complete bipartite, 315
counterclockwise, 307
cycle, 314
declare, 298
default edge kind, 294
default edge operator, 297
edge, 273
edge label, 274
edge label', 274
edge node, 273
edge quotes, 287
edge quotes center, 287
edge quotes mid, 287
edges, 273
empty nodes, 281
every graph, 273
fresh nodes, 278
grid placement, 302
group polar shift, 305
group shift, 300
grow down, 301
grow down sep, 304
grow left, 301
grow left sep, 303
grow right, 301
grow right sep, 303
grow up, 301
grow up sep, 303
induced complete bipartite, 316
induced cycle, 314
induced independent set, 314
induced path, 315
left anchor, 276
m, 312
matching, 316
matching and star, 316
math nodes, 281
multi, 286
n, 311
name, 279
name separator, 279
name shore V, 312
name shore W, 312
new --, 277
new ->, 276
new -!-, 277
new <-, 277
new <->, 277
no placement, 299
nodes, 273
not ⟨*color class name*⟩, 292
number nodes, 279
number nodes sep, 279
operator, 293
path, 315
phase, 306
placement/chain count, 310
placement/compute position, 310
placement/depth, 310
placement/element count, 308
placement/level, 307
placement/logical node depth, 310
placement/logical node width, 309
placement/place, 311
placement/width, 308
put node text on incoming edges, 291
put node text on outgoing edges, 291
quick, 285
radius, 306
recolor ⟨*color class name*⟩ by, 292
right anchor, 277
simple, 286
source edge clear, 290
source edge node, 290
source edge style, 290
target edge clear, 289
target edge node, 289
target edge style, 289
trie, 283
typeset, 280
use existing nodes, 278
V, 311
W, 312
wrap after, 313
x, 299
y, 300
gridded, 579
grow, 336
grow cyclic, 846
grow via three points, 845
grow', 337
growth function, 338
growth parent anchor, 338
handle active characters in code, 576
handle active characters in nodes, 576
height, 833
help lines, 161
huge circuit symbols, 610
huge mindmap, 714
id, 346
if, 587
in, 839
in control, 842

in distance, 842
in front of path, 223
in looseness, 841
in max distance, 842
in min distance, 842
info, 613
info sloped, 614
info', 613
info' sloped, 614
initial, 573
initial above, 573
initial below, 573
initial by arrow, 573
initial by diamond, 573
initial distance, 573, 574
initial left, 573
initial right, 573
initial text, 573
initial where, 573
inner color, 778
inner frame sep, 578
inner frame xsep, 578
inner frame ysep, 578
inputs, 618, 619
insert path, 153
intersection/
  by, 144
  name, 144
  of, 144
  sort by, 145
  total, 144
isometric view, 738
join, 604
jump mark left, 350
jump mark mid, 350
jump mark right, 350
key attribute, 668
label, 249
label distance, 250
label position, 249
large circuit symbols, 610
large mindmap, 714
late options, 260
left, 238, 241
left color, 776
left delimiter, 711
lens, 834
level, 334
level 1 concept, 716
level 2 concept, 716
level 3 concept, 716
level 4 concept, 716
level ⟨*number*⟩, 334
level distance, 335
light emitting, 628
line cap, 174
line join, 174
line to, 839
line width, 173
loop, 842
loop above, 842
loop below, 843
loop left, 843
loop right, 843
loose background, 578
loosely dash dot, 176
loosely dash dot dot, 176
loosely dashed, 176
loosely dotted, 175
looseness, 841
lower left, 777
lower right, 777
magnification, 834
make snapshot after, 414
make snapshot if necessary, 414
make snapshot of, 413
mark, 346
mark indices, 347
mark options, 347
mark phase, 347
mark repeat, 347
mark size, 347
matrix/
  inner style order, 324
  inner style/cell, 324
  inner style/column, 324
  inner style/even odd column, 324
  inner style/even odd row, 324
  inner style/every cell, 324
  inner style/row, 324
matrix, 318
matrix anchor, 326
matrix of math nodes, 710
matrix of nodes, 709
max distance, 841
medium circuit symbols, 610
meet, 850
mid left, 242
mid right, 242
middle color, 776
midway, 247
min distance, 841
mindmap, 713
missing, 337
miter limit, 174
month code, 586
month label above centered, 594
month label above left, 594
month label above right, 595
month label below centered, 595
month label below left, 595
month label left, 593
month label left vertical, 593
month label right, 593
month label right vertical, 593
month list, 592
month text, 586
month xshift, 583
month yshift, 583
move to, 839
name, 131, 153, 224, 358
name intersections, 144
name path, 144
name path global, 144
name prefix, 226
name prefix .., 265

name suffix, 226
near end, 247
near start, 247
nearly opaque, 354
nearly transparent, 354
new set, 282
no markers, 348
no marks, 348
node contents, 223
node distance, 240
node font, 231
node halign header, 235
node quotes mean, 254
nodes, 322
nodes in empty cells, 710
nonzero rule, 180
numbered faces, 725
ohm, 627
on background layer, 577
on chain, 602
on grid, 240
only marks, 352
opacity, 353
opaque, 354
out, 839
out control, 842
out distance, 842
out looseness, 841
out max distance, 842
out min distance, 841
outer color, 779
outer frame sep, 579
outer frame xsep, 579
outer frame ysep, 579
overlay, 258
parabola height, 162
parametric, 345
parent anchor, 340
path fading, 360
path picture, 181
pattern, 179
pattern color, 179
patterns/
  bottom left, 736
  bounding box, 736
  infer tile bounding box, 736
  tile size, 736
  tile transformation, 736
  top right, 736
perspective/
  p, 739
  q, 740
  r, 740
perspective, 739
pic actions, 263
pic text, 265
pic text options, 265
pic type, 262
pics/
  background code, 264
  code, 263
  foreground code, 264
pin, 251
pin distance, 251
pin edge, 251
pin position, 251
place, 744
plane origin, 565
plane x, 565
plane y, 565
point down, 613
point left, 613
point right, 613
point up, 612
polar comb, 351
pos, 244
post, 745
postaction, 188
pre, 745
pre and post, 745
preaction, 186
prefix, 346
prefix after command, 153
quotes mean label, 253
quotes mean pin, 254
radius, 157
range, 345
raw gnuplot, 346
rdf engine/
  get new resource curie, 766
  get scope curie, 767
  prefix, 763
  scope is new context, 767
  statement, 764
  statements/has as member, 768
  statements/has type, 766
  statements/is a bag, 768
  statements/is a container, 768
  statements/is a sequence, 768
  statements/is an alternative, 769
  statements/object, 765
  statements/predicate, 765
  statements/subject, 765
rdf engine, 763
rdf engine on, 763
relationship, 667
relative, 840
remember picture, 258
reset cm, 378
resistor, 626
right, 238, 241
right color, 776
right delimiter, 712
root concept, 716
rotate, 377
rotate around, 377
rotate around x, 377
rotate around y, 378
rotate around z, 378
rotate fit, 685
rounded corners, 156
row ⟨*number*⟩, 323
row ⟨*row number*⟩ column ⟨*column number*⟩, 323
row sep, 321
samples, 343

samples at, 343
save path, 170
scale, 376
scale around, 376
scope fading, 362
semithick, 173
semitransparent, 354
set, 282
shade, 182
shading, 182
shading angle, 182
shadow scale, 780
shadow xshift, 780
shadow yshift, 781
shape, 224
sharp corners, 157
sharp plot, 348
shift, 375
shift only, 375
shorten <, 209
shorten >, 209
show background bottom, 580
show background grid, 578
show background left, 580
show background rectangle, 577
show background right, 580
show background top, 579
sibling angle, 846
sibling distance, 336
size, 833
slice, 851
sloped, 246
small circuit symbols, 610
small mindmap, 714
smooth, 348
smooth cycle, 349
solid, 175
spy connection path, 835
spy scope, 833
spy using outlines, 836
spy using overlays, 836
start angle, 159
start branch, 605
start chain, 600
state, 572
state with output, 572
state without output, 572
step, 160
structured tokens, 747
subgraph nodes, 445
subgraph text bottom, 447
subgraph text none, 446
subgraph text sep, 447
subgraph text top, 446
swap, 245
tension, 348
text, 231
text depth, 236
text effects, 658
text height, 236
text opacity, 355
text width, 233
thick, 173
thin, 173
tight background, 578
tiny circuit symbols, 610
to path, 165
token, 745
token distance, 746
tokens, 746
top color, 776
transform canvas, 379
transform shape, 243
transform shape nonlinear, 244
transition, 744
transparency group, 363
transparent, 353
trim left, 184
trim right, 184
turn, 146
turtle/
  back, 848
  bk, 849
  distance, 848
  fd, 848
  forward, 848
  home, 848
  how, 848
  left, 849
  lt, 849
  right, 849
  rt, 849
turtle, 848
ultra nearly opaque, 354
ultra nearly transparent, 354
ultra thick, 174
ultra thin, 173
upper left, 777
upper right, 777
use as bounding box, 183
use path, 170
variable, 343
very near end, 247
very near start, 247
very nearly opaque, 354
very nearly transparent, 354
very thick, 173
very thin, 173
view, 851
visualizer color, 936
week list, 591
width, 833
x, 373
x radius, 157
xbar, 352
xbar interval, 352
xcomb, 351
xrange, 346
xscale, 376
xshift, 376
xslant, 376
xstep, 160
y, 374
y radius, 157
ybar, 351
ybar interval, 352

ycomb, 351
year code, 587
year text, 587
yrange, 346
yscale, 376
yshift, 376
yslant, 377
ystep, 160
z, 374
tikz/
ac source, 629
adjustable, 630
ampere, 630
amperemeter, 629
and gate, 620
backward diode, 629
battery, 629
break contact, 629
breakdown diode, 629
buffer gate, 620
bulb, 629
capacitor, 629
contact, 629
coulomb, 630
current direction, 629
current direction', 629
current source, 629
data visualization/
scientific polar axes/-180 to 180, 968
scientific polar axes/-90 to 0, 968
scientific polar axes/-90 to 90, 968
scientific polar axes/-pi half to 0, 969
scientific polar axes/-pi half to pi half, 969
scientific polar axes/-pi to pi, 969
scientific polar axes/0 to 180, 968
scientific polar axes/0 to 2pi, 969
scientific polar axes/0 to 360, 968
scientific polar axes/0 to 90, 968
scientific polar axes/0 to pi, 969
scientific polar axes/0 to pi half, 969
scientific polar axes/fourth quadrant, 970
scientific polar axes/fourth quadrant clockwise, 970
scientific polar axes/left half, 971
scientific polar axes/left half clockwise, 971
scientific polar axes/lower half, 970
scientific polar axes/lower half clockwise, 970
scientific polar axes/quadrant, 970
scientific polar axes/quadrant clockwise, 970
scientific polar axes/right half, 971
scientific polar axes/right half clockwise, 971
scientific polar axes/upper half, 970
scientific polar axes/upper half clockwise, 970
dc source, 629
diode, 629
direction info, 630
farad, 630
ground, 629
henry, 630
hertz, 630
inductor, 629
light dependent, 630
light emitting, 630
make contact, 629
nand gate, 620
nor gate, 620
not gate, 620
ohm, 630
ohmmeter, 629
or gate, 620
resistor, 629
Schottky diode, 629
siemens, 630
tunnel diode, 629
volt, 630
voltage source, 629
voltampere, 630
voltmeter, 629
watt, 630
xnor gate, 620
xor gate, 620
Zener diode, 629
tikz package, 128
TikZ math functions
coordinate, 705
for, 706
function, 707
if, 707
int, 705
integer, 705
let, 704
print, 708
real, 705
return, 707
\tikzaliascoordinatesystem, 142
\tikzappendtofigurename, 673
\tikzdeclarecoordinatesystem, 142
\tikzdeclarepattern, 736
\tikzdvdeclarestylesheetcolorseries, 936
\tikzexternaldisable, 675
\tikzexternalenable, 676
\tikzexternalfiledependsonfile, 674
\tikzexternalize, 670
\tikzexternalrealjob, 670
\tikzfading, 359
tikzfadingfrompicture environment, 358, 359
\tikzgraphforeachcolorednode, 294
\tikzgraphnodefullname, 280
\tikzgraphnodename, 280
\tikzgraphnodepath, 280
\tikzgraphnodetext, 280
\tikzgraphpreparecolor, 295
\tikzgraphsset, 273
\tikzifexternalizing, 679
\tikzifexternalizingnext, 679
\tikzinputsegmentfirst, 644
\tikzinputsegmentlast, 644
\tikzinputsegmentsupporta, 644
\tikzinputsegmentsupportb, 644

\tikzlastnode, 260
\tikzmath, 703
tikzpicture environment, 128, 130
\tikzpicturedependsonfile, 673
\tikzrdfhashmark, 764
\tikzset, 133
\tikzsetexternalprefix, 672
\tikzsetfigurename, 673
\tikzsetnextfilename, 672
tile size key, 732, 736
tile transformation key, 732, 736
time key, 386
tiny circuit symbols key, 610
.tip handler, 208
tips key, 190
To arrow tip, 216
to path operation, 163
to path key, 165
token key, 745
token distance key, 746
tokens key, 746
top key, 1082
top color key, 776
top right key, 732, 736
topaths library, 839
topScope (Lua), 560
total key, 144
tpp coordinate system, 738
Transform (Lua), 518
transform (Lua), 516
transform key, 370
transform canvas key, 379
transform shape key, 243
transform shape nonlinear key, 244
Transform.
    concat (Lua), 518
    invert (Lua), 519
    new (Lua), 518
    new_rotation (Lua), 518
    new_scaling (Lua), 518
    new_shift (Lua), 518
transition key, 744
translate basic layer animation attribute, 1222
translate system layer animation attribute, 1262
transparency group key, 363
transparent key, 353
trapezium shape, 788
trapezium angle key, 788
trapezium left angle key, 788
trapezium right angle key, 788
trapezium stretches key, 789
trapezium stretches body key, 789
tree layout key, 458
trees graph drawing library, 458
trees library, 845
Triangle arrow tip, 219
triangle plot mark, 758
Triangle Cap arrow tip, 220
triangle* plot mark, 758
triangles decoration, 651
trie key, 283
trig format key, 1041
trim left key, 184
trim lowlevel key, 185
trim right key, 184
true key, 1000
true math function, 1043
.try handler, 994
Tuesday date test, 1010
tunnel diode key, 629
turn key, 146
Turned Square arrow tip, 219
turtle key, 848
turtle library, 848
two screens with lagging second layout, 1019
two screens with optional second layout, 1019
type key, 731
typeset key, 280

ultra nearly opaque key, 354
ultra nearly transparent key, 354
ultra thick key, 174
ultra thin key, 173
unknown choice value key, 995
unknown key key, 995
unrooted rectangular phylogram key, 486
unrooted straight phylogram key, 487
unshift (Lua), 514
unshiftByCoordinate (Lua), 514
unweighted pair group method using arithmetic averages key, 484
up to date check key, 675
updatePriority (Lua), 538
upgma key, 484
upper left key, 777
upper right key, 777
use as bounding box key, 183
use comma key, 1060
use existing nodes key, 278
use path key, 170
use period key, 1060
use set key, 871
\useasboundingbox, 171
\usegdlibrary, 420
\usepgflibrary, 1077
\usepgfmodule, 1077
\usetikzlibrary, 128
/utils/
    exec, 995
uv axes key, 887
uv Cartesian key, 887
uvw axes key, 887
uvw Cartesian cabinet key, 887

V key, 311
\value, 868
value key, 388
.value forbidden handler, 986
value forbidden key, 995
.value required handler, 986
value required key, 995
var key, 1006
var make contact IEC shape, 636
var resistor IEC shape, 634
variable key, 343
variation key, 448
vary dashing style sheet, 937

vary hue style sheet, 940
vary thickness style sheet, 937
vary thickness and dashing style sheet, 938
veclen math function, 1045
verbatim key, 1062
verbose key, 677
verbose IO key, 677
verbose optimize key, 677
version=⟨version⟩ package option, 1076
Vertex (Lua), 503
Vertex.
    new (Lua), 505
Vertex:
    anchor (Lua), 506
    boundingBox (Lua), 505
vertical key, 430
vertical line through key, 143
vertical lines pattern, 730
vertical' key, 430
very near end key, 247
very near start key, 247
very nearly opaque key, 354
very nearly transparent key, 354
very thick key, 173
very thin key, 173
view animation attribute, 404
view basic layer animation attribute, 1224
view key, 851
viewbox system layer animation attribute, 1264
views library, 850
visibility system layer animation attribute, 1258
visible animation attribute, 395
visible basic layer animation attribute, 1218
visualization coordinate system, 862
visualize as line key, 925
visualize as scatter key, 928
visualize as smooth line key, 926
visualizer color key, 936
visualizer label options/auto key, 942
visualizer label options/index key, 942
visualizer label options/node style key, 943
visualizer label options/pin angle key, 944
visualizer label options/pin length key, 944
visualizer label options/pos key, 942
visualizer label options/text key, 941
visualizer label options/text colored key, 943
visualizer label options/text' key, 941
visualizer label options/when key, 941
visualizer options/gap cycle key, 927
visualizer options/gap line key, 927
visualizer options/ignore style sheets key, 924
visualizer options/label in data key, 941
visualizer options/label in legend key, 946
visualizer options/label in legend options key, 959
visualizer options/no lines key, 927
visualizer options/pin in data key, 943
visualizer options/polygon key, 926
visualizer options/smooth cycle key, 927
visualizer options/smooth line key, 926
visualizer options/straight cycle key, 926
visualizer options/straight line key, 926
visualizer options/style key, 924
volt key, 630
voltage source key, 629
voltampere key, 630
voltmeter key, 629

W key, 312
\w@pgf@writea, 1228
watt key, 630
waves decoration, 643
Wednesday date test, 1010
week list key, 591
weekend date test, 1010
weight key, 448
west fading, 683
when key, 863
whom key, 1211
width key, 193, 833, 1111, 1118
width math function, 1046
width' key, 193
workday date test, 1010
wrap after key, 313

\x, 167
x key, 136, 299, 373, 731, 738, 1083
x plot mark, 757
x radius key, 137, 138, 157
xbar key, 352
xbar interval key, 352
xcomb key, 351
xnor gate key, 620
xnor gate IEC symbol key, 624
xor gate key, 620
xor gate IEC symbol key, 624
xrange key, 346
xscale animation attribute, 399
xscale key, 376
xshift animation attribute, 399
xshift key, 376, 733–735
xskew animation attribute, 399
xskew basic layer animation attribute, 1221
xslant animation attribute, 399
xslant key, 376
xstep key, 160
xy axes key, 887
xy Cartesian key, 887
xy polar coordinate system, 138
xyz coordinate system, 136
xyz axes key, 887
xyz Cartesian cabinet key, 887
xyz cylindrical coordinate system, 564
xyz polar coordinate system, 137
xyz spherical coordinate system, 564

\y, 167
y key, 136, 300, 374, 731, 738, 1083
y radius key, 137, 138, 157
ybar key, 351
ybar interval key, 352
ycomb key, 351
year code key, 587
year text key, 587
yrange key, 346
yscale animation attribute, 399
yscale key, 376

`yshift` animation attribute, 399
`yshift` key, 376, 733–735
`yskew` animation attribute, 399
`yskew` basic layer animation attribute, 1221
`yslant` animation attribute, 399
`yslant` key, 377
`ystep` key, 160

`z` key, 136, 374, 564, 738
`Zener diode` key, 629
`zerofill` key, 1054
`zigzag` decoration, 639