SparseM: A Sparse Matrix Package for R *

Roger Koenker and Pin Ng

January 30, 2026

Abstract

SparseM provides some basic R functionality for linear algebra with sparse matrices. Use of the package is illustrated by a family of linear model fitting functions that implement least squares methods for problems with sparse design matrices. Significant performance improvements in memory utilization and computational speed are possible for applications involving large sparse matrices.

1 Introduction

Many applications in statistics involve large sparse matrices, matrices with a high proportion of zero entries. A typical example from parametric linear regression involves longitudinal data with fixed effects: many indicator variables consisting of a few ones and a large number of zero elements. In nonparametric regression, e.g. smoothing splines design matices are extremely sparse often with less than 1% of nonzero entries. Conventional algorithms for linear algebra in such situations entail exorbitant storage requirements and many wasteful floating point operations involving zero entries. For some specially structured problems, e.g. banded matrices, special algorithms are available. But recent developments in sparse linear algebra have produced efficient methods for handling unstructured sparsity in a remarkably efficient way.

Exploiting these developments, the package SparseM provides some basic linear algebra functionality for sparse matrices stored in several standard formats. The package attempts to make the use of these methods as transparent as possible by adhering to the method-dispatch conventions of *R*.[1] Functions are provided for: coercion, basic unary and binary operations on matrices and linear equation solving.

Our implementation is based on Sparskit (Saad (1994)), which provides one of the more complete collection of subroutines for BLAS like functions and sparse

*This package should be considered experimental. The authors would welcome comments about any aspect of the package. This document is an R vignette prepared with the aid of `Sweave`, Leisch(2002). Support from NSF SES 99-11184 is gratefully acknowledged.

[1]The first release of the SparseM packaged used *S3* method-dispatch, the current release has adopted the new *S4* method dispatch. Our thanks to Brian Ripley and Kurt Hornik for advice on this aspect of the package.

matrix utilities available in the public domain.[2] Our Cholesky factorization and backsolve routines are based on Ng and Peyton (1993), which still appears to represent the state of the art for solving linear systems involving symmetric positive definite matrices.[3]

In Section 2 we discuss in more detail the components of the package, provide some examples on their use and explain the basic design philosopy. Section 3 discusses some refinements proposed for future implementations.

SparseM can be obtained from the Comprehensive R Archive Network, CRAN, at `http://cran.r-project.org/`.

2 Design Philosophy

In this section we briefly describe some aspects of our design philosophy beginning with the question of storage modes.

2.1 Storage Modes

There are currently more than twenty different storage formats used for sparse matrices. Each of these formats is designed to exploit particular features of the matrices that arise in various applications areas to gain efficiency in both memory utilization and computation. Duff, Erisman and Reid (1986) and Saad (1994) provide detailed accounts of the various storage schemes. Following Saad (1994) we have chosen compressed sparse row (**csr**) format as the primary storage mode for SparseM.[4] An n by m matrix A with real elements a_{ij}, stored

[2] Recently, a sparse matrix version of BLAS subprograms has been provided by Duff, Heroux and Pozo (2002). Unfortunately, it handles only sparse matrix times dense matrix multiplication at the Level 3 Sparse BLAS, but not sparse matrix times sparse matrix multiplication. The sparse matrix utilities available in Sparskit, e.g. masking, sorting, permuting, extracting, and filtering, which are not available in Sparse BLAS, are also extrememly valuable. Sparse linear algebra is a rapidly developing field in numerical analysis and we would expect to see many important new developments that could be incorportated into SparseM and related code in the near future.

[3] There are also several new direct methods for solving unsymmetric sparse systems of linear equations over the last decade. A rather comprehensive comparison of performance of some prominent software packages for solving general sparse systems can be found in Gupta (2002). Unfortunately, the comparisons do not include the Peyton and Ng algorithm employed here. The top performer reported in the study is WSMP (Gupta, 2000) which requires proprietary XLF Fortran complier, XLC C compilier and the AIX operating system, and the library is not released under the GPL license. The runner up reported is MUMPS (Amestoy, Duff, L'Excellent and Koster, 2002) which has a non-commerical license but is written in Fortran 90. The third best performer is UMFPACK (Davis, 2002), which is implemented in MATLAB Version 6.0 and later, also has a non-commerical license. Since it is a general sparse solver not written specifically for symmetric positive definite systems of linear equations, it would be interesting to see how it compares with the Choleski factorization of Peyton and Ng adopted here.

[4] Other sparse storage formats supported in SparseM include compressed sparse column (**csc**), symmetric sparse row (**ssr**) and symmetric sparse column (**ssc**). The data structure of **csc** format is the same as that of **csr** format except the information is stored column-wise. The **ssr** and **ssc** formats are special cases of **csr** and **csc**, respectively, for symmetric matrices, only the information in the lower triangle is stored. We have created new class

in **csr** format consists of three arrays:

- `ra`: a real array of nnz elements containing the non-zero elements of A, stored in row order. Thus, if $i < j$, all elements of row i precede elements from row j. The order of elements within the rows is immaterial.
- `ja`: an integer array of nnz elements containing the column indices of the elements stored in `ra`.
- `ia`: an integer array of $n+1$ elements containing pointers to the beginning of each row in the arrays `ra` and `ja`. Thus `ia[i]` indicates the position in the arrays `ra` and `ja` where the ith row begins. The last $(n+1)$st element of `ia` indicates where the $n+1$ row would start, if it existed.

The following commands illustrate typical coercion operations.

```
> library(SparseM)
> a <- rnorm(5*4)
> a[abs(a)<0.7] <- 0
> A <- matrix(a,5,4)
> A

           [,1]       [,2]       [,3]      [,4]
[1,]  0.0000000 -0.7735001 -1.5703592  0.810163
[2,]  0.0000000  0.0000000  0.9066658  0.000000
[3,] -0.8981058  0.0000000  0.0000000  0.759078
[4,]  0.8393254 -1.1891120  0.0000000 -1.172146
[5,] -0.8717972  1.4913375  0.0000000  0.000000

> A.csr <- as.matrix.csr(A)
> A.csr

Sparse matrix of class "matrix.csr" {use 'str(.)' to see the inner structure}:
     [,1]       [,2]       [,3]       [,4]
[1,]          . -0.7735001 -1.5703592  0.8101630
[2,]          .          .  0.9066658          .
[3,] -0.8981058          .          .  0.7590780
[4,]  0.8393254 -1.1891120          . -1.1721455
[5,] -0.8717972  1.4913375          .          .

> as.matrix(A.csr)

           [,1]       [,2]       [,3]      [,4]
[1,]  0.0000000 -0.7735001 -1.5703592  0.810163
[2,]  0.0000000  0.0000000  0.9066658  0.000000
[3,] -0.8981058  0.0000000  0.0000000  0.759078
[4,]  0.8393254 -1.1891120  0.0000000 -1.172146
[5,] -0.8717972  1.4913375  0.0000000  0.000000
```

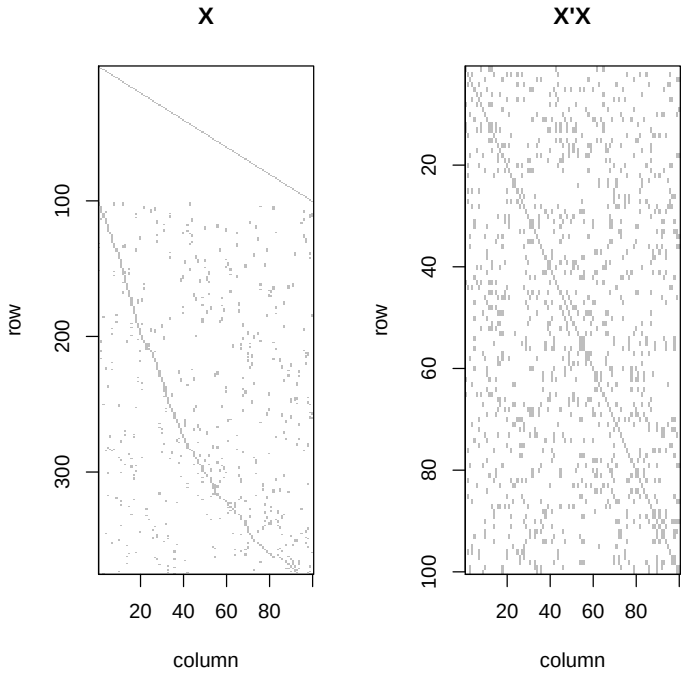
objects, `matrix.csr`, `matrix.csc`, `matrix.ssr`, `matrix.ssc`, for each of these four formats.

>

To facilitate testing we have included `read.matrix.hb` and `write.matrix.hb` to deal with matrices in the Harwell-Boeing storage format. A list of sites with extensive collections of sparse matrices in this format can be found at `http://math.nist.gov/MatrixMarket/`. Details on the Harwell-Boeing format can be found in the help files for `read.matrix.hb` and `write.matrix.hb` as well as in the User's Guide for Harwell-Boeing Sparse Matrix Collection at `ftp://ftp.cerfacs.fr/pub/harwell_boeing/`.

2.2 Visualization

The `image` function allows users to explore the structure of the sparsity in matrices stored in **csr** format. In the next example we illustrate the design matrix for a bivariate spline smoothing problem illustrated in Koenker and Mizera (2002). The upper 100 rows of the matrix are an identity matrix, the lower 275 rows represent the penalty component of the design matrix. In this example X has 1200 nonzero entries, roughly 3.2 percent of the number of floating point numbers needed to represent the matrix in dense form. The $X'X$ form of the matrix has 1162 nonzero elements or 11.62 percent of the entries in the full matrix.



2.3 Indexing and Binding

Indexing and the functions `cbind` and `rbind` for the `matrix.csr` class work just like they do on dense matrices. Objects returned by `cbind` and `rbind` operating on objects of the `matrix.csr` class retain their `matrix.csr` class attribute.

2.4 Linear Algebra

SparseM provides a reasonably complete set of commonly used linear algebra operations for the `matrix.csr` class. The general design philosophy for this set of functions is that operations on `matrix.csr` class will yield an object also in `matrix.csr` class with a few exceptions mentioned below.

The functions `t`, and `%*%` for transposition, and multiplication of **csr** matrices work just like their dense matrix counterparts and the returned objects retain their `matrix.csr` class. The `diag` and `diag<-` functions for extracting and assigning the diagonal elements of **csr** matrices also work like their dense matrix counterparts except that the returned objects from `diag` are dense vectors with appropriate zeros reintroduced. The unary and binary functions in the group generic functions Ops return objects of `matrix.csr` class.

2.5 Linear Equation Solving

Research on solutions to sparse symmetric positive definite systems of linear equations has focused primarily on methods based on the Cholesky factorization, and we have followed this approach. There are three functions `chol`, `backsolve` and `solve` to handle a symmetric positive definite system of linear equations. `chol` performs Cholesky factorization using the block sparse Cholesky algorithms of Ng and Peyton (1993). The result can then be passed on to `backsolve` with a right-hand-side to obtain the solutions. For systems of linear equations that only vary on the right-hand-side, the result from `chol` can be reused, saving considerable computing time. The function `solve`, which combines the use of `chol` and `backsolve`, will compute the inverse of a matrix by default, if the right-hand-side is missing.The data structure of the `chol.matrix.csr` object produced by the sparse Cholesky method is comewhat complicated. Users interested in recovering the Cholesky factor in some more conventional form should recognize that the original matrix has undergone a permutation of its rows and columns before Cholesky factorization; this permutation is given by the `perm` component of the structure. Currently no coercion methods are supplied for the class `chol.matrix.csr`, but the computation of the determinant by extracting the diagonal of the Cholesky factor offers some clues for how such coercion could be done. This determinant is provided as a component of the `chol.matrix.csr` structure because it can be of some value in certain maximum likelihood applications.

2.6 Least Squares Problems

To illustrate the functionality of the package we include an application to least squares regression. The group of functions `slm`, `slm.fit`, `slm.fit.csr`, `summary.slm` and `print.summary.slm` provide analogues of the familiar `lm` family. In the current implementation `slm` processes a formula object in essentially the same way as `lm`, and calls an intermediate function `slm.fit`, which in turn calls `slm.fit.csr` where the actual fitting occurs. Rather than the usual QR decomposition, `slm.fit.csr` proceeds by backsolving the triangular system resulting from a Cholesky decomposition of the $X^\top X$ matrix. The sparsity of the resulting structure is usually well preserved by this strategy. The use of sparse methods is quite transparent in the present `slm` implementation and `summary.slm` with the associated `print.summary.slm` should produce identical output to their cousins in the `lm` family. However, the speed and memory utilization can be quite drammatically improved. In the following problem, which involves a design matrix that is 1850 by 712 there is a nearly three hundred fold improvement in speed (on a Sun Ultra 2) when we compare `lm.fit` and `slm.fit`. The comparison is somewhat less compelling between `lm` and `slm` since there is a substantial common fixed cost to the setup of the problems. In addition to the computational time saved there is also a significant reduction in the memory required for large sparse problems. In extreme cases memory becomes a binding constraint on the feasibility of large problems and sparse storage is critical in expanding the range of problem sizes. This is particularly true of applications in smoothing and related image processing contexts.

```
> #hb.o <- read.matrix.hb(system.file("HBdata","lsq.rra",package = "SparseM"))
> data(lsq)
> X <- model.matrix(lsq) #extract the design matrix
> y <- model.response(lsq) # extract the rhs
> X1 <- as.matrix(X)
> slm.time <- system.time(slm(y~X1-1) -> slm.o) # pretty fast
> lm.time <- system.time(lm(y~X1-1) -> lm.o) # very slow
> slm.fit.time <- system.time(slm.fit(X,y)) # very fast
> lm.fit.time <- system.time(lm.fit(X1,y)) # still very slow
> cat("slm time =",slm.time,"\n")

slm time = 0.031 0.009 0.038 0 0

> cat("lm time =",lm.time,"\n")

lm time = 0.119 0.004 0.123 0 0

> cat("slm.fit time =",slm.fit.time,"\n")

slm.fit time = 0.058 0 0.059 0 0

> cat("lm.fit time =",lm.fit.time,"\n")
```

```
lm.fit time = 0.104 0 0.105 0 0

> cat("slm Results: Reported Coefficients Truncated to 5  ","\n")

slm Results: Reported Coefficients Truncated to 5

> sum.slm <- summary(slm.o)
> sum.slm$coef <- sum.slm$coef[1:5,]
> sum.slm

Call:
slm(formula = y ~ X1 - 1)

Residuals:
       Min         1Q     Median         3Q        Max
-1.952e-01 -1.400e-02  5.329e-15  1.442e-02  1.783e-01

Coefficients:
     Estimate Std. Error t value Pr(>|t|)
[1,] 823.3613     0.1274  6460.4   <2e-16 ***
[2,] 340.1156     0.1711  1987.3   <2e-16 ***
[3,] 472.9760     0.1379  3429.6   <2e-16 ***
[4,] 349.3175     0.1743  2004.0   <2e-16 ***
[5,] 187.5595     0.2100   893.3   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.03789 on 1138 degrees of freedom
Multiple R-Squared:     1,        Adjusted R-squared:     1
F-statistic: 4.504e+07 on 712 and 1138 DF,        p-value:     0

> cat("lm Results: Reported Coefficients Truncated to 5  ","\n")

lm Results: Reported Coefficients Truncated to 5

> sum.lm <- summary(lm.o)
> sum.lm$coefficients <- sum.lm$coefficients[1:5,]
> sum.lm

Call:
lm(formula = y ~ X1 - 1)

Residuals:
     Min       1Q   Median       3Q      Max
-0.19522 -0.01400  0.00000  0.01442  0.17833

Coefficients:
```

```
    Estimate Std. Error t value Pr(>|t|)
X11 823.3613     0.1274  6460.4   <2e-16 ***
X12 340.1156     0.1711  1987.3   <2e-16 ***
X13 472.9760     0.1379  3429.6   <2e-16 ***
X14 349.3175     0.1743  2004.0   <2e-16 ***
X15 187.5595     0.2100   893.3   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.03789 on 1138 degrees of freedom
Multiple R-squared:      1,        Adjusted R-squared:      1
F-statistic: 4.504e+07 on 712 and 1138 DF,  p-value: < 2.2e-16

>
```

3 Some Potential Refinements

There are still many features that could be usefully added to the package. Among these we would especially like to see: `crossprod`, `row`, `col`, code for `eigen`, `svd` would also be desirable, but seems somewhat more problematic. Support for other storage formats might be eventually useful, although **csr, csc, ssr, ssc** formats seem quite sufficient for most purposes. A major improvement in the `slm` implementation would be to replace the line

```
X <- as.matrix.csr(model.matrix(Terms, m, contrasts))
```

which coerces the dense form of the regression design matrix produced by model.matrix into the sparse form. Ideally, this would be done with a special `.csr` form of `model.matrix`, thus obviating the need to construct the dense form of the matrix. We have not looked carefully at the question of implementing this suggestion, but we (still) hope that someone else might be inspired to do so.

Our primary motivation for *R* sparse linear algebra comes from our experience, see e.g. Koenker, Ng and Portnoy (1994) and He and Ng (1999), with interior point algorithms for quantile regression smoothing problems. We plan to report on this experience elsewhere.

References

Amestoy, P. R., I. S. Duff, J. -Y. L'Excellent and J. Koster. (2002). MUltifrontal Massively Parallel Solver (MUMPS Version 4.2 beta) Users' Guide, http://www.enseeiht.fr/lima/apo/MUMPS/

Davis, T. A. (2002). UMFPACK Version 4.0 User Guide, http://www.cise.ufl.edu/research/sparse/umfpack.

Duff, I.S., A. M. Erisman and J. K. Reid. (1986). *Direct Methods for Sparse Matrices*, Clarendon Press, Oxford.

Duff, I. S., M. A. Heroux, and R. Pozo. (2002). "An Overview of the Sparse Basic Linear Algebra Subroutines: The New Standard from the BLAS Technical Forum," *ACM Transactions on Mathematical Software*, 28, 239-267.

Gupta, A. (2000). WSMP: Watson Sparse Matrix Package (Part-II: direct solution of general sparse systems). Technical Report RC 21888 (98472), IBM T.J. Watson Research Center, Yorktown Heights, N.Y., http://www.cs.umn.edu/~agupta/doc/wssmp-paper.ps

Gupta, A. (2002). "Recent Advances in Direct Methods for Solving Unsymmetric Sparse Systems of Linear Equations," *ACM Transactions on Mathematical Software*, 28, 301-324.

He, X., and P. Ng (1999): "COBS: Qualitatively Constrained Smoothing Via Linear Programming," *Computational Statistics*, 14, 315–337.

Koenker, R., P. Ng, and S. Portnoy (1994): "Quantile smoothing splines," *Biometrika*, 81, 673–680.

Leisch, F. (2002). Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis, `http://www.wu-wien.ac.at/am`.

Koenker, R. and Mizera, I (2002). Penalized Triograms: Total Variation Regularization for Bivariate Smoothing, preprint.

Ng, E. G. and B. W. Peyton. (1993) Block sparse Cholesky algorithms on advanced uniprocessor computers", *SIAM J. Sci. Comput.*, 14, 1034-1056.

Saad, Y. (1994) Sparskit: A basic tool kit for sparse matrix computations; Version 2, `http://www.cs.umn.edu/Research/arpa/SPARSKIT/sparskit.html`