Maxwell Equations on S^2 - Implementation

A Semester Thesis on Boundary Element Equations for Maxwell-Type Problems on the Sphere

Oded Stein, ETH Zürich

Supervised by:
Prof. Ralf Hiptmair, ETH Zürich
Prof. Stefan Kurz, Tampere University of Technology
Elke Spindler, ETH Zürich

24. September, 2014

Abstract

My bachelor's thesis 'Maxwell Equations on S^2' [8] focused on the theoretical part of formulating a boundary element method for the solution of the following Maxwell-type problem:

$$(\delta d - k^2)u = 0 \tag{0.1}$$

on the sphere S^2 where u is a zero or one form defined on a simply connected $\Omega \subset S^2$ and the boundary conditions are given on the boundary Γ of a simply connected $C \subset S^2$. The bachelor's thesis [8] is heavily based on previous research by Auchmann and Kurz [5].

This semester thesis is concerned with the actual details of implementation of the boundary element method. the implementation is done in C and MATLAB using MEX [2], building on previous code by Patrick Meury [6].

The numerical analysis shows that the case where the boundary condition is constant can be simulated quite well. The case of the general boundary condition can be solved qualitatively correctly, but there is some error due to the implementation of the hypergeometric function. A better implementation of the hypergeometric function might improve the situation.

Acknowledgements

I would like to thank my supervisors, Prof. Ralf Hiptmair from ETH Zurich and Prof. Stefan Kurz from Tampere University of Technology who already advised me on my bachelor's thesis for helping me with the implementation where necessary and suggest testing methods for the implementation.

I would also like to thank Elke Spindler who introduced me to Patrick Meury's 2D-BEM code and helped me with the countless bugs that turned up in the programming process.

Contents

1 Introduction

The intent of the bachelor's thesis [8] was to solve the following Maxwell-type problem on a subset of the sphere S^2:

$$(\delta d - k^2)u = 0 \qquad \text{in } \Omega \subseteq S^2 \tag{1.1}$$

where u is a differential form and boundary conditions are given on $\Gamma = \partial\Omega$, simply connected. The problem is then solved using indirect boundary element methods for the single-layer potential.

Because of several difficulties obtaining a kernel to use in the single layer potential, the problem is solved using a kernel for the Helmholtz problem.

For one-forms, [8] gives the following Helmholtz kernel:

$$w_1(s) = \frac{\Gamma\left(\frac{3-\kappa}{2}\right)\Gamma\left(\frac{3+\kappa}{2}\right)}{8\pi}\left(1-\sin^2\frac{s}{2}\right)F\left(\frac{3-\kappa}{2},\frac{3+\kappa}{2},3,1-\sin^2\frac{s}{2}\right) \tag{1.2}$$

For zero-forms the following Helmholtz kernel is given:

$$w_0(s) = -\frac{1}{4\sin\left(\pi\frac{\kappa-1}{2}\right)}P_{\frac{\kappa-1}{2}}\left(2\sin^2\frac{s}{2}-1\right) = -\frac{1}{4\sin\left(\pi\frac{\kappa-1}{2}\right)}F\left(\frac{1-\kappa}{2},\frac{1+\kappa}{2},1,1-\sin^2\frac{s}{2}\right) \tag{1.3}$$

Using a lemma, they can be used to formulate the single layer potential:

$$\tilde{\Psi}_p\omega := \left(\Psi_p - \frac{1}{k^2}d\Psi_{p-1}\delta\right)\omega \tag{1.4}$$

And to go with it, the single layer operator:

$$\tilde{V}_p := \operatorname{tr}\tilde{\Psi}_p \tag{1.5}$$

These are then used to formulate the following weak integral equation:

$$\langle\tilde{V}_1\omega,\eta\rangle_\Gamma = \langle\beta,\eta\rangle_\Gamma \tag{1.6}$$

where $\beta \in H_\perp^{-\frac{1}{2}}\Omega^p(d,\Gamma)$ is the Dirichlet boundary condition and $\omega \in H_\parallel^{-\frac{1}{2}}\Omega^p(\delta,\Gamma)$ and $\eta \in H_\parallel^{-\frac{1}{2}}\Omega^p(\delta,\Gamma)$ is the discretization space.

Once solved for ω, the solution can be recovered with the following representation formula for $x \notin \Gamma$:

$$u(x)[v_x] = (\tilde{\Psi}_p \omega)[v_x] \tag{1.7}$$

In [8] the problem then is discretized using piecewise differentiable functions (hat functions) in the following way:

$$\varphi_i(\xi) := \begin{cases} 0 & \text{if } \xi \notin [x_{i-1}, x_{i+1}) \\ \frac{\xi - x_{i-1}}{h_i} & \text{if } \xi \in [x_{i-1}, x_i) \\ \frac{x_{i+1} - \xi}{h_{i+1}} & \text{if } \xi \in [x_i, x_{i+1}) \end{cases} \tag{1.8a}$$

with respective derivative:

$$\Phi(\xi) := \varphi_i'(\xi) = \begin{cases} 0 & \text{if } \xi \notin [x_{i-1}, x_{i+1}) \\ \frac{1}{h_i} & \text{if } \xi \in [x_{i-1}, x_i) \\ -\frac{1}{h_{i+1}} & \text{if } \xi \in [x_i, x_{i+1}) \end{cases} \tag{1.8b}$$

Equation (1.6) then becomes the following Galerkin problem:

$$\sum_i \omega_i^{(N)} \left(K_{i,j} + L_{i,j}\right) = R_j \tag{1.9}$$

$$\begin{aligned} K_{i,j} &:= \int_0^d \int_0^d \varphi_i(\zeta)\varphi_j(\xi) w_1(s(\chi(\xi), \chi(\zeta))) g(dl^\sharp(\chi(\xi)), P_x^y dl^\sharp(\chi(\zeta)))\, d\zeta d\xi \\ L_{i,j} &:= -\frac{1}{k^2} \int_0^d \int_0^d w_0(s(\chi(\xi), \chi(\zeta)))\Phi_i(\zeta)\Phi_j(\xi)\, d\zeta d\xi \\ R_j &:= \int_0^d b(\chi(\xi))\varphi_j(\xi)\, d\zeta \end{aligned} \tag{1.10}$$

The representation formula is given by:

$$u^\sharp(x)[v_x] = \sum_i \omega_i^{(N)} P_i(v_x) \tag{1.11}$$

$$\begin{aligned} P_i(v_x) = \int_0^d & w_1(s(\chi(x), \chi(y)))\varphi_i(\zeta) g(v_x, \operatorname{tr} P_x^y dl^\sharp(\chi(\zeta))) \\ & + \frac{1}{k^2} dw_0(s(\chi(x), \chi(y)))[v_x]\Phi_i(\zeta) d\zeta \end{aligned} \tag{1.12}$$

2 Structure of the Program

The program is built in MATLAB and split into multiple MEX files that are called by a master script. The structure can be seen in figure 2.

To run the program, first the `startup` script must be run. Then a boundary mesh `c` from the folder `Meshes` has to be selected (a $3 \times n$ matrix) and, depending on it, boundary conditions `dir` (a function handle) from the folder `BoundaryConditions`.

In the next step a single layer matrix `V` is assembled with the command `assem_SingleLayer_Maxwell` (which uses `c`) and a load vector `L` is assembled with the command `assem_LoadVector` (which uses `c` and `dir`). Those are used to calculate the boundary density with `omega = V\L`.

Now the potential has to be calculated. This is done by setting up a $3 \times n$ matrix `points` with points where the potential should be evaluated[1] (those are usually generated with something like MATLAB's `sphere` command). This is then passed to `potEval_Maxwell` (which also uses `c` and `omega`) to get a vector proxy of the solution at each point in `points`.

Those were all MATLAB scripts or MEX files that make up the interface of the programs. The MEX files themselves rely on several helper C files that are present in the folder `C_files`. They will be discussed in the following chapters.

[1] Care has to be taken when selecting these points: they should not be too close to the boundary as of singularities in the potential evaluation, and they should be on the right side of the boundary.

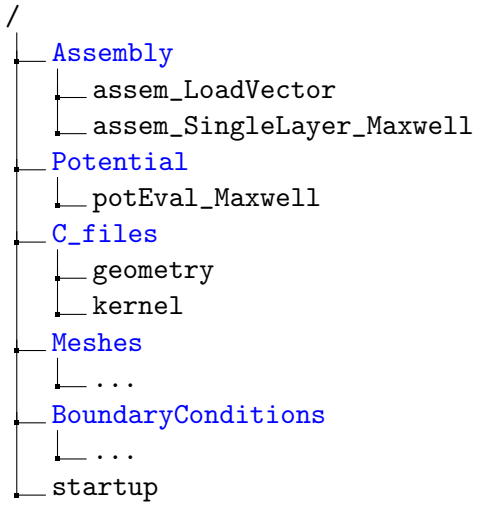

Figure 2.1: The structure of the implementation in MATLAB. Directories are marked in blue.

3 Implementation Details

In this chapter the interesting implementation details of the scripts from chapter 2 will be discussed.

3.1 assem_LoadVector

This MEX program computes the load vector for an input consisting of a coordinate mesh (the boundary) and a MATLAB function handle (the Dirichlet condition). It is a loop over all boundary elements calculating the load vector from the discretization introduced in [8] and mentioned in equation 1.9.

The actual quadrature is done with the one-dimensional Gauss-Legendre quadrature from [6]. The only interesting code here is the parametrization of the boundary. This can't be done the way Patrick Meury did it in his 2D setting, but must be adapted to the sphere embedded in $\mathbb{R}^3$. An embedding of the following form is used, where p_0 is the beginning of the arc integrated over and p_1 is the end:

$$\omega(\varphi) = p_0 \cos\varphi + \frac{p_0 \times p_1 \times p_0}{|p_0 \times p_1 \times p_0|} \sin\varphi \tag{3.1}$$

The functional determinant of this is simply the arc length. This can be seen in code snippet 5.1.

3.2 assem_SingleLayer_Maxwell

This MEX program computes the matrix corresponding to the discretized single layer operator for an input consisting of a coordinate mesh (the boundary). It is a double loop over all boundary elements calculating $K_{i,j}$ and $L_{i,j}$ from equation 1.9. It is a double loop over all boundary elements.

There are three different quadrature types used in calculating the double integral. They are all directly taken from [6] or slightly modified versions from [6].
If the two panels of the double integral overlap, [6]'s composite Gauss-Legendre for double integrals with logarithmic singularity on the diagonal is used. It can be seen in code snippet 5.2
If the two panels of the double integral coincide on one end, [6]'s composite Gauss-Legendre for double integrals with a singularity in the corner is used. It can be seen

in code snippet 5.3. To make sure the singularity is always at the correct corner, the orientation of some panels has to be reversed depending on where the two panels meet. If the two panels of the double integral are distant panels, a normal Gauss-Legendre-type quadrature is used.

The embedding is the same as in `assem_LoadVector`.
The implementation of the kernel function is discussed in the section `kernel`.

3.3 potEval_Maxwell

This MEX program evaluates the potential at a number of input points, given a coordinate mesh (the boundary) and an appropriate boundary density ω. It is an integral over all boundary elements calculating (1.11).

The quadrature and embedding are the same as with `assem_LoadVector`. This is also the reason the chosen points can't be too close to the boundary, otherwise the quadrature method can't handle the singularity.

The implementations of w_1 and dw_0 are explained in the section about `kernel`.

3.4 geometry

This is a C helper file called by all other files. The most important routines in here calculate geodesic distance (code snippet 5.4) and parallel transport (code snippet 5.5) (as described in [8] as well as finding a tangent vector on a geodesic (code snippet 5.6).

3.5 kernel

This is a C helper file that contains math-heavy functions such as all the kernel functions and a routine to calculate the hypergeometric function. It is called by the MEX files. Some implementations here use routines from the GNU scientific library (GSL) [1].

The Helmholtz w_0 and w_1 kernels are implemented as in (1.2) and (1.3). The exact codes can be seen in code snippets 5.7 and 5.8.

They call the hypergeometric function, unless the argument is small: then a logarithmic approximation of the singularity is used. The logarithmic approximation is calculated using the fact that near $z = 1$, the hypergeometric function has the following behavior [4]:

$$F(a, b, a + b, z) \sim \frac{\Gamma(a + b)}{\Gamma(a)\Gamma(b)} \left(\log(1 - z) + \psi(a) + \psi(b) + 2\gamma\right) \tag{3.2}$$

where ψ is the digamma function, γ is the Euler-Mascheroni constant. The exact code can be seen in code snippet 5.9.

The exterior derivative dw_0 has not been calculated in [8], but is used in `kernel`. The derivation of the exterior derivative can be seen in section 5.2.1.

Throughout the `kernel` file the hypergeometric function `F` is used multiple times. As GSL's implementation didn't perform well due to bugs, the single fraction approach from [7] was used to implement the hypergeometric function. For small arguments of s (about $\lesssim 0.1$) a logarithmic and polynomial expansion around the pole is used. The expansion is taken from [4].

4 Numerical Results

In this chapter the numerical results of the method are presented and discussed.

4.1 Constant boundary conditions along an equator

For the case where the boundary condition is constant tangent along the boundary and the boundary is the equator, we have an exact solution. The exact solution and its derivation can be found in section 5.2.2.

Pictures of the solution can be seen in figure 4.1.

The error was evaluated on two points, $p = \frac{1}{\sqrt{3}}(-1,-1,-1)$. The convergence diagram can be seen in figure 4.1. It can be seen that the convergence is a bit worse than linear. This is because, for the constant boundary case, the method computes the boundary density exactly (remember: curved boundary elements are used unlike in the traditional $\mathbb{R}^3$ BEM), so the result is already very accurate. The only error is introduced by the quadrature and not by the BEM. This error causes the small convergence.

4.2 General boundary conditions along some boundary

For the case of testing general boundary condition, recall that the kernel of the Maxwell equation (mentioned in [8]'s (4.3)) is a solution in any area if we pick some $y \in S^2$ and $v_y \in T_y S^2$ and exclude an open area around y. Let the complement of that open area be Ω and its boundary Γ. Then if the projection of the kernel on Γ is used as Dirichlet condition, the method should reproduce the kernel inside Ω.

The process above applied to two different boundaries can be seen in figure 4.2. The method computes the same solution no matter how the boundary is chosen, which is what should be expected.

The error here was evaluated at $p = \frac{1}{\sqrt{3}}(-1,-1,-1)$. The convergence plot can be seen in figure 4.2. At the moment there is no real convergence and quite an error. This is probably due to the highly unstable implementation of the hypergeometric function. This problem worsens the smaller $c-a-b$ is, and as this is evaluated multiple times in the calculation of the kernel these errors accumulate.

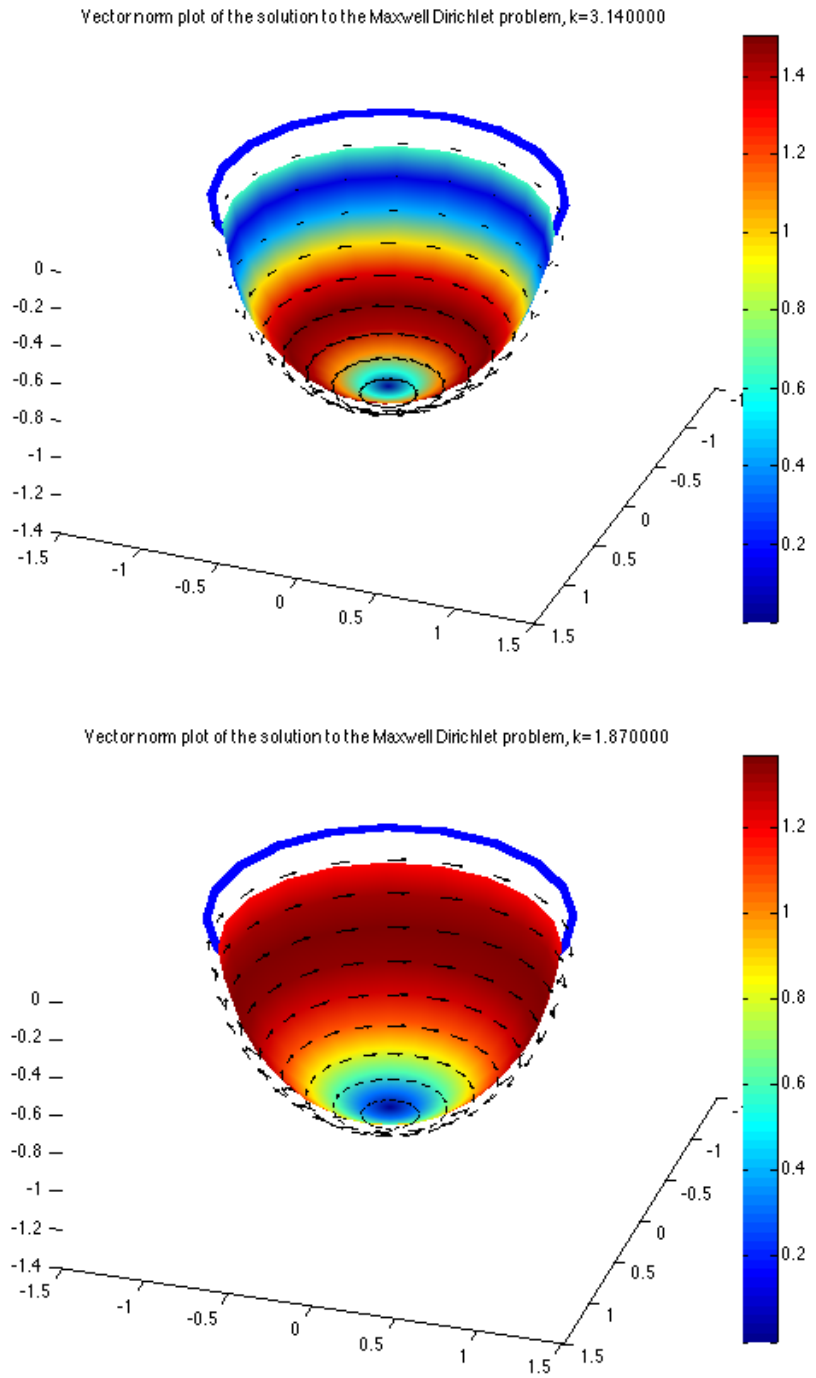

Figure 4.1: The plot of the vector proxy to the solution of the Maxwell problem with constant Dirichlet boundary at the equator (thick blue line). The coloring represents the norm of the vectors, the arrows the actual vectors. The first picture is for $k = 3.14$, the second picture for $k = 1.87$

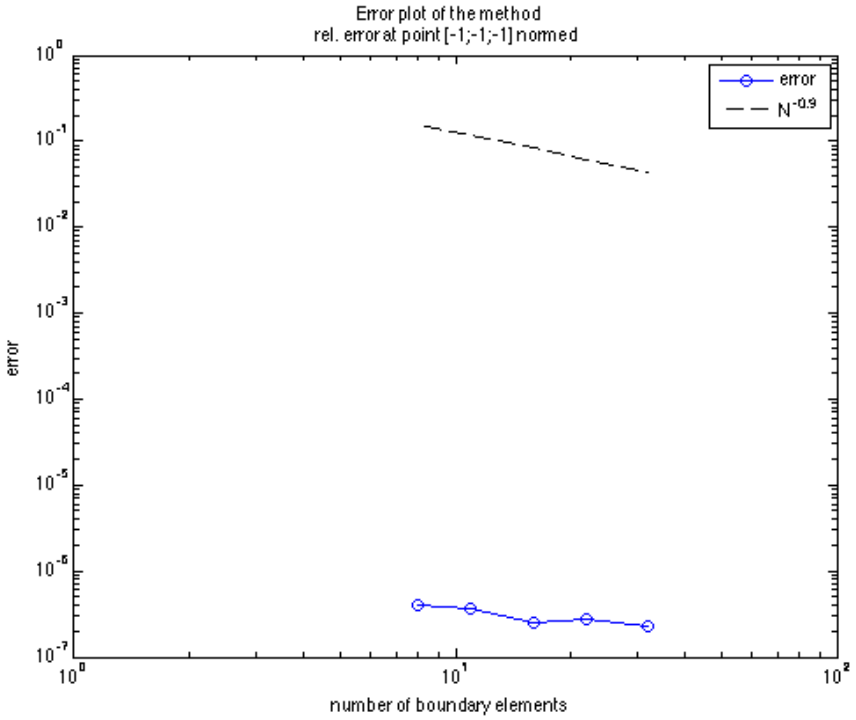

Figure 4.2: Convergence plots of the method for constant boundary conditions on the equator evaluated at $p = \frac{1}{\sqrt{3}}(-1,-1,-1)$. The picture is for $k = 3.14$.

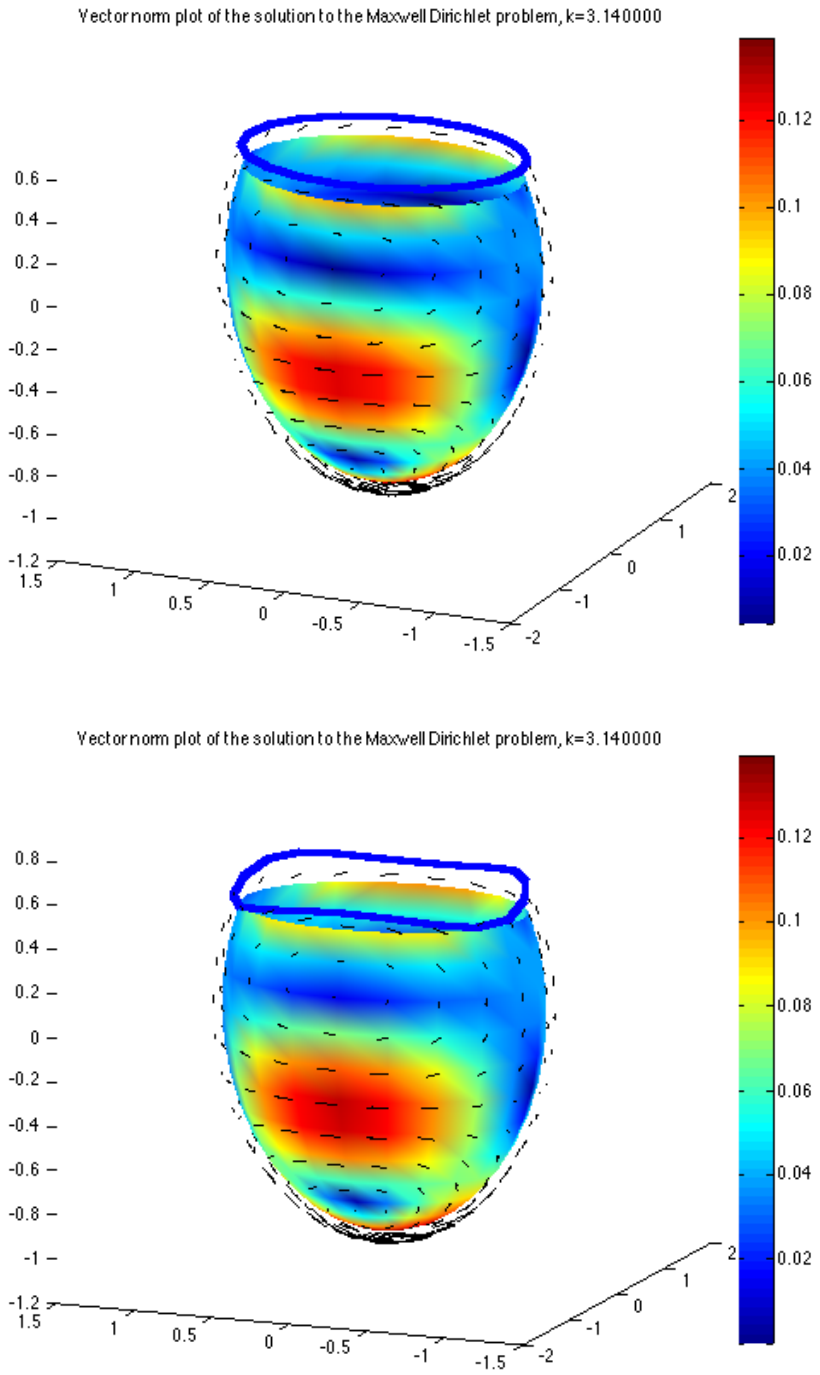

Figure 4.3: The plot of the vector proxy to the solution of the Maxwell problem with general boundary conditions at two different boundaries. The pictures are for $k = 3.14$.

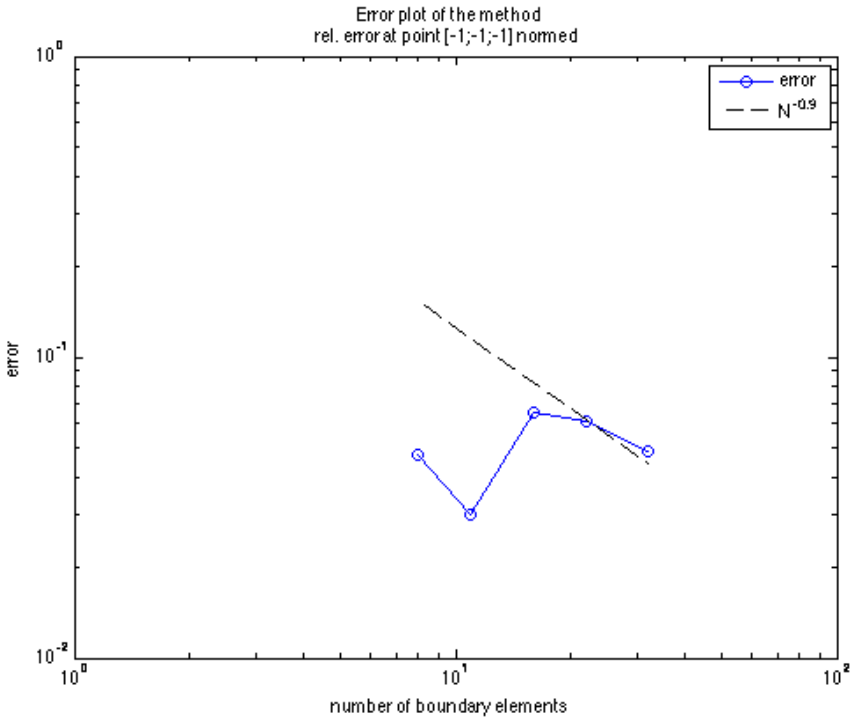

Figure 4.4: Convergence plots of the method for general boundary conditions evaluated at $p = \frac{1}{\sqrt{3}}(-1,-1,-1)$. The boundary is the equator. The picture is for $k = 3.14$.

The solution seen in 4.2 is however qualitatively correct. This becomes clear if the solution is correctly scaled at a reference point and then the convergence is evaluated. This can be seen in figure 4.2 where this has been done: at one reference point the scaling was determined, then at three other points (pt1, pt2, pt3) the relative error was measured. In this case we get extremely quick convergence.

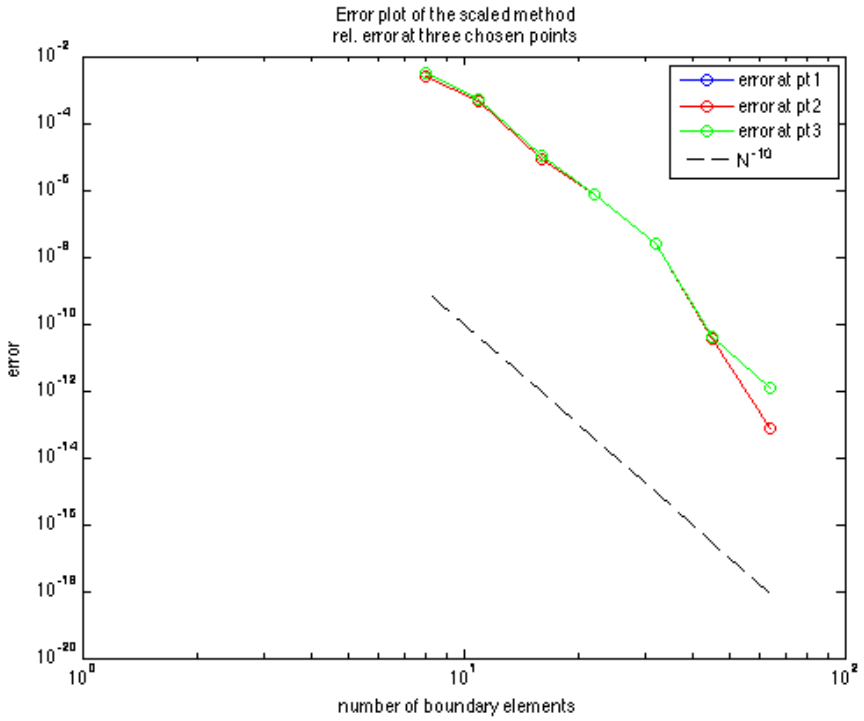

Figure 4.5: Convergence plots of the scaled method (scaled at some reference point of the calculated solution) for general boundary conditions evaluated at three different points. The boundary is the equator. The picture is for $k = 3.14$.

5 Appendix

5.1 Code

The actual code is not included with this appendix, but was submitted with the semester thesis. Interested readers can request it via e-mail to `steino@student.ethz.ch`.

Only special code excerpts referred to elsewhere in the text will be listed here. They are not included in the other chapters for the sake of readability.
Some implementations here use routines from the GNU scientific library [1].

Code Snippet 5.1: Embedding used to calculate the integral over arcs on the sphere. panel_start is a vector to the beginning of the panel, panel_end is a vector to the end of the panel and get_tangent_vector(a,b,c) calculates a tangent vector on the panel from b to c based at a.

```
double *const panel_tangent_vector =
    get_tangent_vector(panel_start, panel_start, panel_end);

for(int cntr=0; cntr<n_quadrature_points; ++cntr) {
    /* Find position of i-th quadrature point on straight panel */
    double x[3];
    double t = h*quadrature_points[cntr];
    x[0] = panel_start[0]*cos(t) + panel_tangent_vector[0]*sin(t);
    x[1] = panel_start[1]*cos(t) + panel_tangent_vector[1]*sin(t);
    x[2] = panel_start[2]*cos(t) + panel_tangent_vector[2]*sin(t);

}
free(panel_tangent_vector);
```

Code Snippet 5.2: [6]'s code to generate quadrature points and weights for a 2D quadrature with singularity on the diagonal. `x2` and `w2` contain points and weights for the composite Gauss-Legendre quadrature. `qr->x_i`, `qr->y_i` and `qr->w_i` end up containing x and y quadrature points and quadrature weights for the 2D quadrature respectively.

```
ii = 0;
for(i=0; i<q2; i++){
  wxi = w2[i];
  xxi = x2[i];
  for(j=0; j<q2; j++){
    weta = w2[j];
    xeta = x2[j];

    mxAssert(ii >= 0 && ii < q,"Index out of bounds");
    qr->w_i[ii] = xeta*weta*wxi;
    qr->x_i[ii] = xeta*(1.0-xxi);
    qr->y_i[ii] = xeta;
    ii++;

    mxAssert(ii >= 0 && ii < q,"Index out of bounds");
    qr->w_i[ii] = xeta*weta*wxi;
    qr->x_i[ii] = xeta;
    qr->y_i[ii] = xeta*(1.0-xxi);
    ii++;

  }
}
```

Code Snippet 5.3: [6]'s code to generate quadrature points and weights for a 2D quadrature with singularity in one corner. `x2` and `w2` contain points and weights for the composite Gauss-Legendre quadrature. `qr->x_i`, `qr->y_i` and `qr->w_i` end up containing x and y quadrature points and quadrature weights for the 2D quadrature respectively.

```
ii = 0;
for(i=0; i<q2; i++){
  wxi = w2[i];
  xxi = x2[i];
  for(j=0; j<q2; j++){
    weta = w2[j];
    xeta = x2[j];

    mxAssert(ii >= 0 && ii < q,"Index out of bounds");
    qr->w_s[ii] = xxi*weta*wxi;
    qr->x_s[ii] = xxi;
    qr->y_s[ii] = xeta*xxi;
```

```
      ii++;

      mxAssert(ii >= 0 && ii < q,"Index out of bounds");
      qr->w_s[ii] = xxi*weta*wxi;
      qr->x_s[ii] = xxi*xeta;
      qr->y_s[ii] = xxi;
      ii++;

    }
  }
```

Code Snippet 5.4: The code used to calculate the sin of the geodesic distance between `x` and `y`, divided by 2.

```
  double d0 = x[0]-y[0];
  double d1 = x[1]-y[1];
  double d2 = x[2]-y[2];

  double d = sqrt(d0*d0 + d1*d1 + d2*d2);

  return d/2;
```

Code Snippet 5.5: The code used to calculate parallel transport of the vector `wy` at `y` to `x`.

```
  if(x[0]==y[0] && x[1]==y[1] && x[2]==y[2]) {
      /* No actual transport is happening here.
        Return the same vector. */
      double *vx = malloc(3*sizeof(double));
      vx[0] = wy[0];
      vx[1] = wy[1];
      vx[2] = wy[2];

      return vx;
  }

  double *a = cross_product(y,x);
```

```
if(a[0]==0 && a[1]==0 && a[2]==0) {
    /* x and y have parallel position vectors. It is ok to set a
      to anything as long as it is perpendicular to one of them. */
    double someVector1[] = {1.0, 0.0, 0.0};
    double someVector2[] = {0.0, 1.0, 0.0};

    free(a);
    a = cross_product(x,someVector1);

    if(a[0]==0 && a[1]==0 && a[2]==0) {
        free(a);
        a = cross_product(x,someVector2);
    }
}

double norma = norm_3d(a);
a[0] = a[0]/norma; a[1] = a[1]/norma; a[2] = a[2]/norma;

double *by = cross_product(a, y);
double *bx = cross_product(a, x);

double gwyby = riemann_product(wy,by);
double gwya = riemann_product(wy,a);

double *vx = malloc(3*sizeof(double));
vx[0] = gwyby*bx[0] + gwya*a[0];
vx[1] = gwyby*bx[1] + gwya*a[1];
vx[2] = gwyby*bx[2] + gwya*a[2];

free(bx);
free(by);
free(a);

return vx;
```

Code Snippet 5.6: The code used to generate a tangent vector on the geodesic from `panel_start` to `panel_end` at `x`

```
/* Get a tangent vector in the plane spanned by the angle from
  panel_start to panel_end */
double *upward_vector = cross_product(panel_start, panel_end);
```

```
//Some exception handling omitted

//Get a vector perpendicular to x and the upward normal vector
double *unnormalized_tangent_vector = cross_product(upward_vector,x);
double *tangent_vector = malloc(3*sizeof(double));

double tnorm = norm_3d(unnormalized_tangent_vector);
tangent_vector[0] = unnormalized_tangent_vector[0]/tnorm;
tangent_vector[1] = unnormalized_tangent_vector[1]/tnorm;
tangent_vector[2] = unnormalized_tangent_vector[2]/tnorm;

//Memory management
free(unnormalized_tangent_vector);
free(upward_vector);

return tangent_vector;
```

Code Snippet 5.7: The code used to calculate w_0 at the points `x`, `y`

```
double kappa = sqrt(1 + 4*k*k);

double sins = sin_geodesic_distance_d2(x,y);

double C = -1/(4*sin(M_PI*(kappa-1)/2));
double nu = (kappa-1)/2;

double Ped = F(-nu, nu+1, 1, 1 - sins*sins);

if(isnan(Ped)) {
    /* We are too close to a singularity:
      Try logarithmic approximation */
    Ped = sing_P(nu, sins*sins);
}

return C*Ped;
```

Code Snippet 5.8: The code used to calculate w_1 at the points `x`, `y`

```
double kappa = sqrt(1 + 4*k*k);
```

```
double sins = sin_geodesic_distance_d2(x,y);

double C = gsl_sf_gamma((3-kappa)/2)*
  gsl_sf_gamma((3+kappa)/2)/(8*M_PI);
double a = (3-kappa)/2;
double b = (3+kappa)/2;

double Fed = F(a, b, 3, 1-sins*sins);

if(isnan(Fed)) {
    /* We are too close to a singularity:
      Try logarithmic approximation */
    Fed = sing_hypergeom(a, b, 3, sins*sins);
}

return C*(1-sins*sins)*Fed;
```

Code Snippet 5.9: The code used to calculate dw_0 at the points `x`, `y` with input vector `vx`

```
double kappa = sqrt(1 + 4*k*k);

double sins = sin_geodesic_distance_d2(x,y);
double coss = sqrt(1-sins*sins);

double C = -1/(4*sin(M_PI*(kappa-1)/2));

double a = (3-kappa)/2;
double b = (3+kappa)/2;
double c = 2;

double scalar_part = C*(kappa*kappa-1)/4*sins*coss*F(a, b, c, 1-sins*sins);
double *vector_part = get_tangent_vector(x,y,x);

double retVal = scalar_part*riemann_product(vx,vector_part);

free(vector_part);
return retVal;
```

5.2 Calculations

Some important calculations are not worked out in the other chapters to make them more readable. They are presented here in detail.

5.2.1 Derivation of dw_0

The following calculation describes the derivation of the exterior derivative dw_0. To calculation is done in Fermi coordinates, this means spherical coordinates where the geodesic between y and x is part of the equator. In these coordinates, if the frame is (s,t) such that s runs along the equator and t is orthogonal to s, w_0 becomes independent of t:

$$w_0(x,y) = CP_{\frac{\kappa-1}{2}}\left(2\sin^2\frac{s}{2}-1\right) \tag{5.1}$$

Let's first rewrite w_0 to make calculation simpler:

$$w_0(s) = CP_{\frac{\kappa-1}{2}}\left(2\sin^2\frac{s}{2}-1\right) = CF\left(\frac{1-\kappa}{2},\frac{1+\kappa}{2};1;1-\sin^2\frac{s}{2}\right) \tag{5.2}$$

with F the hypergeometric function.

By Wolfram function reference [3], the differentiation of the hypergeometric function works as following:

$$\frac{\partial F(a,b;c;z)}{\partial z} = \frac{ab}{c}F(a+1,b+1;c+1;z) \tag{5.3}$$

Then the exterior derivative is simply:

$$\begin{aligned} &C\,d\left(F\left(\frac{1-\kappa}{2},\frac{1+\kappa}{2};1;1-\sin^2\frac{s}{2}\right)\right) \\ &\quad = C\frac{\kappa^2-1}{4}\left(\sin\frac{s}{2}\cos\frac{s}{2}\right)F\left(\frac{3-\kappa}{2},\frac{3+\kappa}{2};2;1-\sin^2\frac{s}{2}\right)ds \end{aligned} \tag{5.4}$$

5.2.2 Computation of the exact solution to constant boundary conditions on the equator

We want to calculate the covector potential caused by unit Dirichlet boundary conditions of the following PDE:

$$(\delta d - k^2)\mathbf{A} = 0 \tag{5.5}$$

We pass to standard spherical coordinates where φ is the longitude and θ is the colatitude. Note first, as the vector potential is tangent to the current, the theta component of $\mathbf{A}$ will be 0. Because of symmetry along the equator, clearly also the φ-part of $\mathbf{A}$ will be independent of φ. We can write $\mathbf{A} = Ad\varphi$.
The equation thus simplifies to:

$$\delta d(Ad\varphi) - k^2 Ad\varphi = 0, \qquad \frac{\partial A}{\partial \varphi} = 0 \tag{5.6}$$

With boundary conditions:

$$A = 1 \text{ for } \theta = \frac{\pi}{2} \tag{5.7}$$

With the usual spherical Riemann metric $\sin^2\theta\, d\varphi^2 + d\theta^2$.

To be able to easily compute the Hodge star, change into coordinates where the metric is the identity matrix by introducing $\tilde{\varphi} = (\sin\theta)\varphi$, $\frac{\partial}{\partial\tilde{\varphi}} = \frac{1}{\sin\theta}\frac{\partial}{\partial\varphi}$, $d\varphi = \frac{1}{\sin\theta}d\tilde{\varphi}$.
The outer derivative applied directly to $Ad\varphi$ is:

$$d(Ad\varphi) = A_\theta d\theta \wedge d\varphi = -A_\theta d\varphi \wedge d\theta \tag{5.8}$$

Again we switch into $\tilde{\varphi}$ coordinates.

$$\begin{aligned}
\star d(Ad\varphi) &= - \star (A_\theta d\varphi \wedge d\theta) = - \star \left(\frac{A_\theta}{\sin\theta} d\tilde{\varphi} \wedge d\theta \right) = -\frac{A_\theta}{\sin\theta} \\
d \star d(Ad\varphi) &= -d\frac{A_\theta}{\sin\theta} = -\frac{\partial}{\partial\theta}\frac{A_\theta}{\sin\theta} d\theta = -\left(-\frac{\cos\theta}{\sin^2\theta} A_\theta + \frac{A_{\theta\theta}}{\sin\theta} \right) d\theta \\
\delta d(Ad\varphi) &= - \star d \star d(Ad\varphi) = -\left(-\frac{\cos\theta}{\sin^2\theta} A_\theta + \frac{A_{\theta\theta}}{\sin\theta} \right) d\tilde{\varphi} = \left(\frac{\cos\theta}{\sin\theta} A_\theta - A_{\theta\theta} \right) d\varphi
\end{aligned} \tag{5.9}$$

This gives the following ODE to solve:

$$\begin{aligned}
\frac{1}{\sin\theta} (\cos\theta A_\theta - \sin\theta A_{\theta\theta}) - k^2 A = 0 \\
\cos\theta A_\theta - \sin\theta A_{\theta\theta} - k^2 \sin\theta A = 0
\end{aligned} \tag{5.10}$$

As of Mathematica, the general solution to this is:

$$A(\theta) = cF\left(\frac{-1-\kappa}{4}, \frac{-1+\kappa}{4}, \frac{1}{2}, \cos^2\theta \right) + d(\cos\theta) F\left(\frac{1-\kappa}{4}, \frac{1+\kappa}{4}, \frac{3}{2}, \cos^2\theta \right) \tag{5.11}$$

Where F is the hypergeometric function and $\kappa = \sqrt{1+4k^2}$. We know this has to be 0 at the poles, so

$$\begin{aligned} A(0) &= cF\left(\frac{-1-\kappa}{4}, \frac{-1+\kappa}{4}, \frac{1}{2}, 1\right) + dF\left(\frac{1-\kappa}{4}, \frac{1+\kappa}{4}, \frac{3}{2}, 1\right) \\ &= c\frac{\sqrt{\pi}}{\Gamma\left(\frac{3}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{3}{4}+\frac{\kappa}{4}\right)} + d\frac{\sqrt{\pi}}{2\Gamma\left(\frac{5}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{5}{4}+\frac{\kappa}{4}\right)} = 0 \\ 0 &= c2\Gamma\left(\frac{5}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{5}{4}+\frac{\kappa}{4}\right) + d\Gamma\left(\frac{3}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{3}{4}+\frac{\kappa}{4}\right) \\ d &= -c\frac{2\Gamma\left(\frac{5}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{5}{4}+\frac{\kappa}{4}\right)}{\Gamma\left(\frac{3}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{3}{4}+\frac{\kappa}{4}\right)} \end{aligned} \tag{5.12}$$

Here the fact was used that $F(a,b,c,1) = \frac{\Gamma(c)\Gamma(c-a-b)}{\Gamma(c-a)\Gamma(c-b)}$.

Thus the solution thus is of the form:

$$\begin{aligned} A(\theta) =& cF\left(\frac{-1-\kappa}{4}, \frac{-1+\kappa}{4}, \frac{1}{2}, \cos^2\theta\right) \\ & - c\frac{2\Gamma\left(\frac{5}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{5}{4}+\frac{\kappa}{4}\right)}{\Gamma\left(\frac{3}{4}-\frac{\kappa}{4}\right)\Gamma\left(\frac{3}{4}+\frac{\kappa}{4}\right)}(\cos\theta)F\left(\frac{1-\kappa}{4}, \frac{1+\kappa}{4}, \frac{3}{2}, \cos^2\theta\right) \quad \text{for } \theta \leq \frac{\pi}{2} \end{aligned} \tag{5.13}$$

Now it is time to insert the boundary condition, which is $A(\frac{\pi}{2}) = 1$. This gives the following equation to solve:

$$1 = c\,(1-0) \tag{5.14}$$

And thus the solution is (5.13) with $c = 1$.

6 Bibliography

[1] GNU. *The GNU Scientific Library*. URL: http://www.gnu.org/software/gsl/ (visited on 10/06/2014).

[2] The MathWorks Inc. *Description of MATLAB's MEX*. URL: http://www.mathworks.ch/ch/help/matlab/ref/mex.html (visited on 10/06/2014).

[3] Wolfram Research Inc. *Derivatives of the hypergeometric function 2F1*. URL: http://functions.wolfram.com/HypergeometricFunctions/Hypergeometric2F1/20/01/05/ (visited on 10/06/2014).

[4] Wolfram Research Inc. *Series expansions for the hypergeometric function 2F1*. URL: http://functions.wolfram.com/HypergeometricFunctions/Hypergeometric2F1/06/01/04/01/02/ (visited on 10/06/2014).

[5] Stefan Kurz and Bernhard Auchmann. "Differential Forms and Boundary Integral Equations for Maxwell-Type Problems". In: *Fast Boundary Element Methods in Engineering and Industrial Applications*. Ed. by Ulrich Langer et al. Vol. 63. Lecture Notes in Applied and Computational Mechanics. 2012, pp. 1–62.

[6] Patrick Meury. "Stable Finite Element Boundary Element Galerkin Schemes for Acoustic and Electromagnetic Scattering". PhD thesis. ETH Zurich, 2007.

[7] John Pearson. "Computation of Hypergeometric Functions". MA thesis. University of Oxford, 2009. URL: http://people.maths.ox.ac.uk/porterm/research/pearson_final.pdf.

[8] Oded Stein. "Maxwell Equations on S2". In: *Bachelor Thesis* (2013).