Master Thesis

# Calderon Preconditioning for Higher Order Boundary Element Method

Yulia Smirnova

11.07.2013

## Contents

# 1 Introduction

Previously a dual Calderon preconditioner based on [3] was created in BETL library [4] for scalar constant discontinuous basis.

Now we want to broaden the application of the dual meshes and dual preconditioners by first creating a hierarchical scalar preconditioner based on the existing dual Calderon preconditioner (Sec.2). The hierarchical structure allows effective work with BEM matrices evaluated on higher order bases.

Secondly a similar dual Calderon preconditioner for lowest order vector functions based on [1] was implemented in Sec.4.

And finally by analogy we constructed a similar hierarchical preconditioner for the vector case ( Sec.5). To this end a higher order vector basis was created and tested (Sec.3).

Throughout this work we limit ourselves with 3-nodal triangular meshes. The similar structures can be created for quadrilaterals as well as for curved geometries.

# 2 Hierarchical Preconditioner for Scalar Bases

## 2.1 General Hierarchical Preconditioner Formula

Consider a variational problem defined on the Hilbert space $H$ written in the abstract form:

$$\text{find } u \in H \text{ such that } a(u,v) = (f,v) \ \ \forall v \in H, \tag{2.1.1}$$

where $a(\cdot,\cdot)$ is a continuous bilinear form on $H$ and $f \in H$ .

Introducing a conformal boundary element discretization $H_h \subset H$ and a piecewise polynomial basis of $H_h$, the discrete variational problem becomes:

$$\text{find } u_h \in H_h \text{ such that } a(u_h,v_h) = (f,v_h) \ \ \forall v_h \in H_h. \tag{2.1.2}$$

Index $h$ stands for the discrete spaces and matrices.

After choosing a basis $\{\phi_i\}_{i=0}^N$ in $H_h$ we obtain from (2.1.2) a system of linear equations given by

$$A_h \mathbf{u}_h = \mathbf{b}_h \tag{2.1.3}$$

where $\mathbf{u}_h$ is the vector of unknowns and $\mathbf{b}_h$ is the right hand side vector.

Imagine that $H_h$ is a space of polynomials of degree $k \geq 1$. Then $H_h$ can be decomposed into a constant part $H_c$ and a complementary part $H_s$, which is of the order $k$, i.e :

$$\begin{aligned} &H_h = H_c \oplus H_s, \\ &\forall u_h \in H_h \quad u_h = u_c + u_s, \text{ where } u_c \in H_c \text{ and } \ u_s \in H_s. \end{aligned} \tag{2.1.4}$$

This space-splitting induces us to choose a special basis of $H_h$, which constitutes a union of the basis functions of $H_c$ – $\{\psi_i^c\}_{i=0}^{N_c}$ and the basis functions of $H_s$ – $\{\psi_i^s\}_{i=0}^{N_s}$:

$$\{\hat{\phi}_i\}_{i=0}^N = \{\psi_i^c\}_{i=0}^{N_c} \cup \{\psi_i^s\}_{i=0}^{N_s}. \tag{2.1.5}$$

In this new basis $\{\hat{\phi}_i\}_{i=0}^N$ the original system matrix $A_h$ of (2.1.3) acquires a $2 \times 2$ block structure:

$$\begin{pmatrix} A_{cc} & A_{cs} \\ A_{sc} & A_{ss} \end{pmatrix} \begin{pmatrix} \mathbf{u}_c \\ \mathbf{u}_s \end{pmatrix} = \begin{pmatrix} \mathbf{b}_c \\ \mathbf{b}_s \end{pmatrix} \tag{2.1.6}$$

In case of boundary elements discretization, when $A(\cdot,\cdot)$ is the single layer potential, all problems hide in the constant part, thus the matrix $A_{cc}$ needs

special preconditioning. Whereas for the surplus part $A_{ss}$ a simple (Jacobi-like) preconditioner would be sufficient. The idea is inspired by a similar approach to linear systems arising from finite element discretization: the lowest order part requires a special preconditioner, the higher order parts allow local treatment. And this approach results in the concept of a block preconditioner, which we suggest constructing in the following way:

$$P^{-1} = \begin{pmatrix} P_{cc}^{-1} & 0 \\ 0 & J_{ss}^{-1} \end{pmatrix}, \tag{2.1.7}$$

where $P_{cc}^{-1}$ is a special preconditioner for the constant part and $J_{ss}^{-1}$ is a Jacobi-like preconditioner for the surplus part.

The splitting (2.1.4) into constant and correction parts is a particular case. The approach is more general and similar constructions could be implemented for any arbitrary splitting $H_h = H_1 \oplus H_2$.

## 2.2 Hierarchical Splitting in Case of Nodal Basis Functions

In the scalar case the space $H = H^{1/2}(\Gamma)$ and $H_h$ is the space spanned by piecewise polynomial discontinuous functions of order $k$ denoted as $S_k$. The basis functions of $S_k$ in BETL are the standard interpolatory nodal basis functions. Such bases do not form a hierarchical sequence, i.e. $\{\phi_i^k\}_{i=0}^{N_k} \not\subset \{\phi_i^{k+1}\}_{i=0}^{N_{k+1}}$, where $\{\phi_i^k\}_{i=0}^{N_k}$ are the nodal basis functions of $S_k$.

In case of hierarchical bases the splitting of the system matrix $A_h$ into block structure (2.1.6) could be simply accomplished via reordering of the functions. Whereas in the case of nodal basis functions a special pre-treatment of the basis is needed to fulfill the splitting (2.1.4). And to this end we create a special matrix T.

Let's consider one triangle. Denote the basis functions corresponding to this triangle by $\{\varphi_{loc_i}\}_{i=0}^{n} \subset H_h$. We want to switch to the new basis functions $\{\widehat{\varphi}_{loc_i}\}_{i=0}^{n} \subset H_h$, such that $\widehat{\varphi}_{loc_0} = 1$. This is a simple linear transformation $T_{loc}$:

$$\begin{aligned} &T_{loc} : \{\widehat{\varphi}_{loc_i}\}_{i=0}^{n} \to \{\varphi_{loc_i}\}_{i=0}^{n} \\ &\widehat{\varphi}_{loc_i} = T_{loc}^T \varphi_{loc_i} \text{ for } i = 1, 2, ..., n. \end{aligned} \tag{2.2.1}$$

On the Fig.1 an example of the matrix $T_{loc}$ for 1D hat functions is presented.

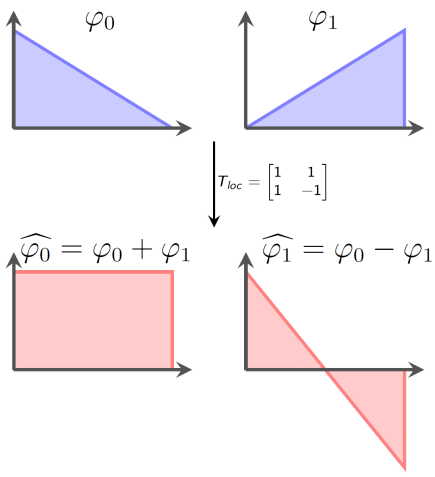



Figure 1: $T_{loc}$ matrix in 1D case

We can preform this transformation on every triangle. In the case of discontinuous basis of $H_h$ basis functions only exist on one triangle and thus they split up into independent groups of $n$ functions. Hence we can apply $T_{loc}$ transformation independently on each triangle of the mesh, and thus the global $T$ matrix would be a block diagonal matrix with $T_{loc}$ blocks on the diagonal:

$$T = \begin{pmatrix} T_{loc} & & \\ & \ddots & \\ & & T_{loc} \end{pmatrix}. \tag{2.2.2}$$

Then any bilinear form $A(\cdot, \cdot)$ will change in the following way

$$A := (A(\varphi_i, \varphi_j))_{i,j=1}^N \to \widehat{A} := (A(\widehat{\varphi_i}, \widehat{\varphi_j}))_{i,j=1}^N = T^T A T \tag{2.2.3}$$

After this transformation the matrix $\widehat{A}$ obtains the desired block structure (2.1.6).

## 2.3 Hierarchical Preconditioner Formula for Bases of Nodal Basis Functions

Now let's return to our system (2.1.3): $A_h u_h = b_h$.

The matrix $T$ (2.2.2) is a linear transformation and our system matrix $A$ would change following the bilinear form transformation formula, i.e.

$$A_H = T^T A T \Rightarrow V = T^{-T} V_H T^{-1}, \tag{2.3.1}$$

where index $H$ stands for 'hierarchical' and matrix $A_H$ has a $2 \times 2$ block structure (2.1.6):

$$A_H = \begin{pmatrix} A_{cc} & A_{cs} \\ A_{sc} & A_{ss} \end{pmatrix} \tag{2.3.2}$$

Thus we can rewrite (2.1.3) as

$$T^{-T} A_H T^{-1} x = b \tag{2.3.3}$$

or in preconditioned form

$$P^{-1} T^{-T} A_H T^{-1} x = b^* = P^{-1} b \tag{2.3.4}$$

Let's choose the preconditioner in the following way

$$P^{-1} = T P_H^{-1} T^T = T \begin{pmatrix} P_{cc}^{-1} & 0 \\ 0 & J_{ss}^{-1} \end{pmatrix} T^T \tag{2.3.5}$$

so that with the change of basis it changes almost like $A$, i.e. as a bilinear form.

Then the final formula for the system is

$$T P_H^{-1} T^T T^{-T} A_H T^{-1} x = T P_H^{-1} A_H T^{-1} x = b^* \tag{2.3.6}$$

We can see that our choice of preconditioner (2.3.5) leads to a reasonable system (2.3.6). Going from right to left we see that

- first matrix $T^{-1}$ transforms $x$ to the hierarchical basis,
- then the system with the matrix and preconditioner both in the hierarchical block form – $A_H$ and $P_H^{-1}$ – is solved,
- and finally matrix $T$ transforms our solution back from the hierarchical basis to the original.

If the system matrix $A_h$ and the preconditioner for the lowest order part $P_{cc}^{-1}$ are symmetric, then the preconditioner (2.3.5) is also symmetric by construction. Moreover it does not depend on the choice of the basis functions in non-constant part of the hierarchical basis, i.e. the choice of the remaining $(n-1)$ rows of the matrix $T$.

The lowest order preconditioner matrix $P_{cc}^{-1}$ is never computed, but realized using the BETL preconditioner routine.

## 2.4 Implementation

The final strategy of the construction of the preconditioner is the following:

1. calculate $T_{loc}$ matrices for the given basis of $H_h$;
2. create the global $T$ matrix from the identical local blocks $T_{loc}$;
3. create a routine that will extract lowest order (constant) basis functions;
4. create inverse block diagonal preconditioner for the surplus part;
5. assemble all these matrices into one preconditioner structure.

### 2.4.1 Projection Matrix $P_c$

The purpose of this matrix is to expand the lowest order preconditioner matrix $B_{cc}$, which has the size $n = \dim H_c$ to the size of the original matrix that we precondition, i.e to $N = \dim H_h$ by adding zero rows and columns. Thus $P_c$ is a $n \times N$ sparse matrix, that has the following structure:

$$(P_c)_{ij} = \begin{cases} 1 & \text{if } j\text{'s dof in } H_h \text{ corresponds to } i\text{'s dof in } H_c \\ 0 & \text{otherwise.} \end{cases} \tag{2.4.1}$$

### 2.4.2 Matrix T Local

As described above the matrix $T_{loc}$ maps hat basis functions of the given order into hierarchical basis (2.2.1). Below this matrix is given for two cases: linear and quadratic hat functions on the reference triangle:

$$\hat{T} := \{(\hat{x}_1, \hat{x}_2) : 0 < \hat{x}_2 < \hat{x}_1 < 1\}. \tag{2.4.2}$$

On this triangle in linear scalar case we have the following basis functions

$$\begin{array}{lllllll} \varphi_0 = \lambda_0 = & 1 & - & \hat{x}_1 & , \\ \varphi_1 = \lambda_1 = & & & \hat{x}_1 & - & \hat{x}_2 & , \\ \varphi_2 = \lambda_2 = & & & & & \hat{x}_2 & , \end{array} \tag{2.4.3}$$

where $\lambda_i, i = 0, 1, 2$ – are the barycentric coordinates on the triangle.

Thus we get that

$$1 = \varphi_0 + \varphi_1 + \varphi_2. \tag{2.4.4}$$

And hence the matrix $T$ has the following structure:

$$T = \begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 1 \\ 1 & 0 & -1 \end{pmatrix}. \tag{2.4.5}$$

In quadratic case the original basis functions used in BETL are

$$\begin{array}{rrl}
\varphi_0 = & \lambda_0(2\lambda_0 - 1) & = 1 - 3\hat{x}_1 + 2\hat{x}_1^2, \\
\varphi_1 = & \lambda_1(2\lambda_1 - 1) & = -\hat{x}_1 + 2\hat{x}_1^2 + \hat{x}_2 - 4\hat{x}_1\hat{x}_2 + 2\hat{x}_2^2, \\
\varphi_2 = & \lambda_2(2\lambda_2 - 1) & = -\hat{x}_2 + 2\hat{x}_2^2, \\
\varphi_3 = & 4\lambda_0\lambda_1 & = 4\hat{x}_1 - 4\hat{x}_1^2 - 4\hat{x}_2 + 4\hat{x}_1\hat{x}_2, \\
\varphi_4 = & 4\lambda_1\lambda_2 & = 4\hat{x}_1\hat{x}_2 - 4\hat{x}_2^2, \\
\varphi_5 = & 4\lambda_2\lambda_0 & = 4\hat{x}_2 - 4\hat{x}_1\hat{x}_2.
\end{array} \tag{2.4.6}$$

We still have that

$$1 = \sum_{i=0}^{5} \varphi_i, \tag{2.4.7}$$

and thus we choose the following $T$:

$$T = \begin{pmatrix}
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 & -1 & -1 \\
1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}. \tag{2.4.8}$$

For each new basis $H_h$ the matrix $T_{loc}$ should be evaluated manually.

### 2.4.3 Inverse Block Diagonal Submatrix

The block diagonal preconditioner is built in the following way:

- from the matrix $A$, that we precondition, extract $D_{ss,loc}$ – $n \times n$ diagonal blocks corresponding to each element, where $n$ is the number of basis functions on a triangle;
- evaluate the product $T_{loc} D_{ss,loc} T_{loc}^T := \widehat{D}_{ss,loc}$ for each element;
- invert the lower right $(n-1) \times (n-1)$ subblock of $\widehat{D}_{ss,loc}$ using *Lapack* routine and zero out the remaining entries, thus creating $\widehat{D}^{-1}_{ss,loc}$;
- construct the global matrix $D_{ss}^{-1}$ from $\widehat{D}^{-1}_{ss,loc}$, placing them back on the diagonal.

$$\widehat{D}_{ss,loc} = \begin{pmatrix} * & * \\ * & D_{sub} \end{pmatrix} \to \widehat{D}^{-1}_{ss,loc} = \begin{pmatrix} 0 & 0 \\ 0 & D^{-1}_{sub} \end{pmatrix} \tag{2.4.9}$$

Extracting the blocks $D_{ss,loc}$ from the original $A$ matrix is very expensive, that's why a simple recalculation of the diagonal blocks was implemented.

Moreover the *Lapack* inversion is only carried out once at the construction step. Altogether $D_{ss}^{-1}$ is a BETL-sparse matrix.

#### 2.4.4 Assembling matrices

Finally the preconditioner is constructed using the following formula:

$$P^{-1} = T(P_c{}^T B_{cc} P_c + D_{ss}^{-1})T^T \tag{2.4.10}$$

where

$B_{cc}$ – is the lowest order preconditioner;

$T$ – is the matrix that divides local basis functions into constant and non-constant parts;

$P_c$ – extracts the constant part of the basis;

$D_{ss}^{-1}$ – is inverse block diagonal preconditioner for the remaining part.

From this formula one can clearly see, that the choice of the basis functions in $H_s = H_h \setminus H_c$ does not matter. Since $D_{ss,loc}^{-1} = (T_{loc} D_{ss_l oc} T_{loc}^T)^{-1}$, and thus for the second summand in (2.4.10) we have just $D_{ss,loc}^{-1}$, matrix $T$ cancels out.

### 2.5 Results

#### 2.5.1 Validation Model

We are testing our preconditioner on the first kind boundary integral equation arising from a Dirichlet problem for the Laplace operator:

$$V_h u_N = \left(\frac{1}{2} M_h + K_h\right) u_D \tag{2.5.1}$$

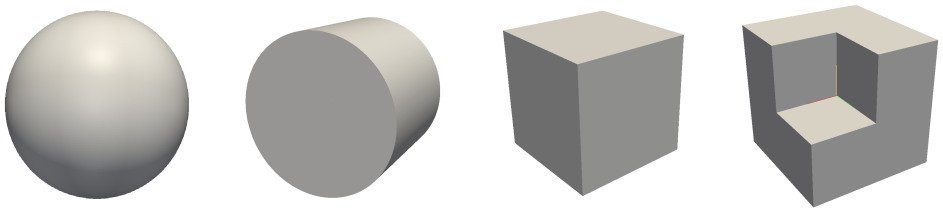
$V_h$ – is the Galerkin matrix of the Laplace single layer potential evaluated on discontinuous scalar functions of order $k$,

$K_h$ – is the Galerkin matrix of the Laplace double layer potential matrix evaluated on discontinuous scalar function of order $k$ and continuous scalar function of order $k+1$,

$M_h$ – is the mass matrix between discontinuous scalar function of order $k$ and continuous scalar function of order $k+1$.

Knowing $u_D$ we solve (2.5.1) and then compare the solution against $u_N$ to validate it.

First we perform these tests on simple geometries: sphere, cylinder, cube and Fichera corner.



(a) sphere (b) cylinder (c) cube (d) Fichera corner

Figure 2: Test geometry

As a preconditioner for the constant part two existing preconditioners are used: Dual Calderon [3] and ABPX [8]. We make tests for linear and quadratic scalar bases, i.e. for $k = 1, 2$ using default BETL ACA settings. Resulting numbers of iterations are presented in Sec.2.5.2, 2.5.3 and 2.5.4.

#### 2.5.2 Test Models: Constant Basis

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 128 | 25 | 24 | 10 | 25 | 24 | 10 |
| 512 | 34 | 28 | 10 | 34 | 28 | 10 |
| 2048 | 43 | 32 | 10 | 43 | 32 | 10 |
| 8192 | 53 | 36 | 10 | 53 | 36 | 10 |
| 32768 | 66 | 40 | 11 | 66 | 40 | 11 |

Table 1: constant basis for $V_h$, sphere

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 254 | 38 | 25 | 13 | 38 | 25 | 13 |
| 1016 | 53 | 31 | 15 | 53 | 31 | 15 |
| 4064 | 85 | 35 | 18 | 85 | 35 | 18 |
| 16256 | 149 | 42 | 20 | 149 | 42 | 20 |

Table 2: constant basis for $V_h$, cylinder

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 384 | 16 | 23 | 7 | 16 | 23 | 7 |
| 1536 | 26 | 28 | 8 | 26 | 28 | 8 |
| 6144 | 33 | 31 | 8 | 33 | 31 | 8 |
| 24576 | 41 | 35 | 8 | 41 | 35 | 8 |

Table 3: constant basis for $V_h$, cube

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 144 | 45 | 29 | 13 | 45 | 29 | 13 |
| 576 | 62 | 36 | 14 | 62 | 36 | 14 |
| 2304 | 80 | 42 | 15 | 80 | 42 | 15 |
| 9216 | 100 | 52 | 16 | 100 | 52 | 16 |
| 36864 | 124 | 53 | 17 | 124 | 53 | 17 |

Table 4: constant basis for $V_h$, Fichera corner

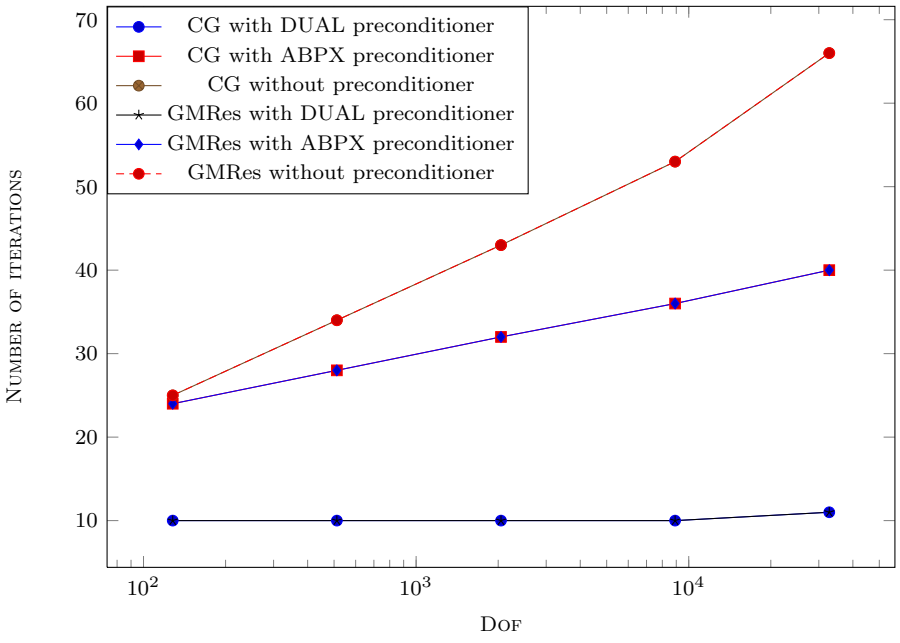



Figure 3: constant basis for $V_h$, sphere

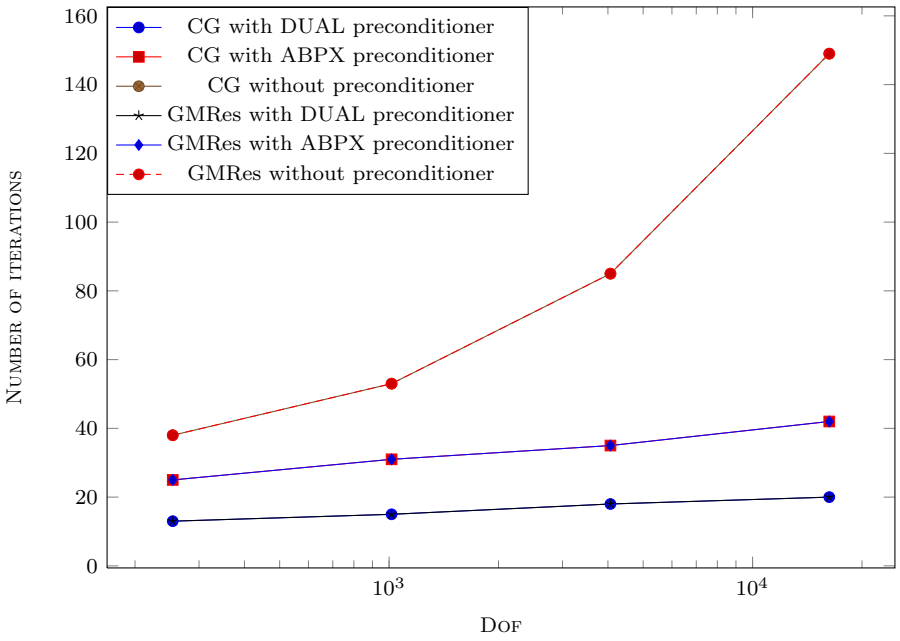



Figure 4: constant basis for $V_h$, cylinder

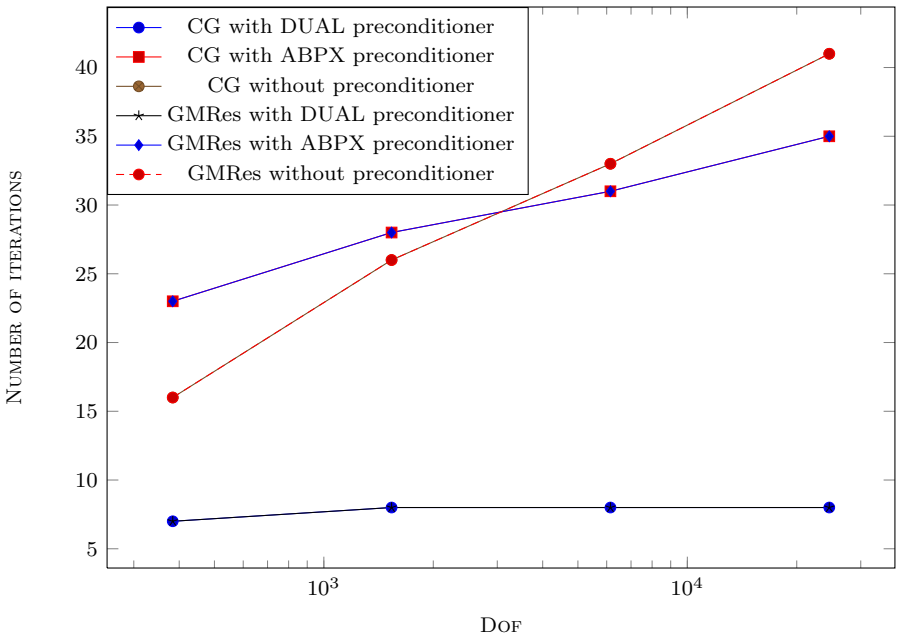



Figure 5: constant basis for $V_h$, cube

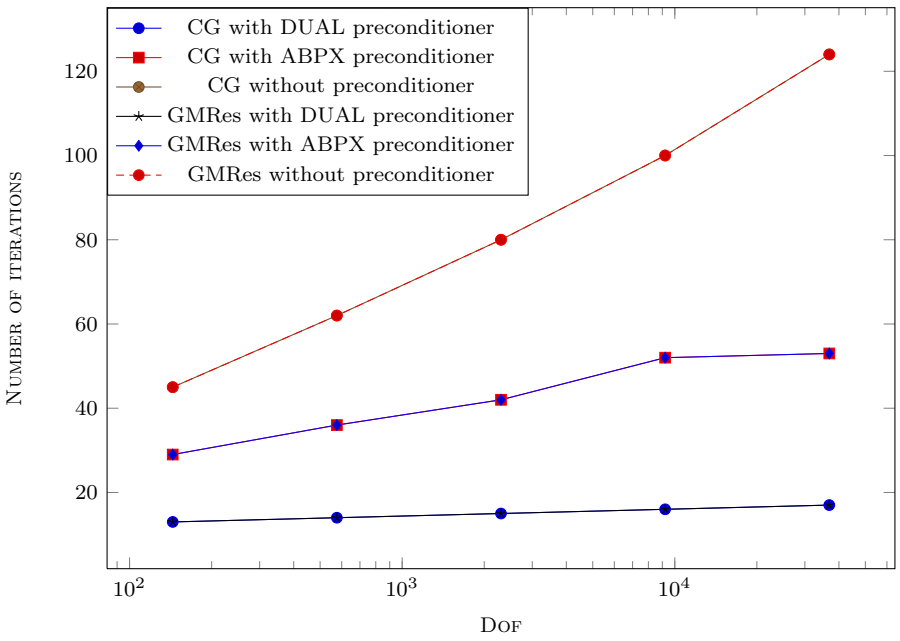



Figure 6: constant basis for $V_h$, Fichera corner

### 2.5.3 Test Models: Linear Basis

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 384 | 49 | 56 | 22 | 59 | 61 | 23 |
| 1536 | 62 | 64 | 23 | 78 | 70 | 25 |
| 6144 | 77 | 71 | 23 | 104 | 77 | 26 |
| 24576 | 97 | 80 | 23 | 144 | 85 | 26 |
| 98304 | 130 | 84 | 24 | 205 | 89 | 28 |

Table 5: linear basis for $V_h$, sphere

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 762 | 73 | 63 | 28 | 96 | 69 | 30 |
| 3048 | 102 | 73 | 32 | 146 | 79 | 36 |
| 12192 | 182 | 84 | 37 | 306 | 91 | 41 |
| 48768 | 388 | 115 | 46 | 778 | 121 | 62 |

Table 6: linear basis for $V_h$, cylinder

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 1152 | 42 | 60 | 20 | 46 | 62 | 19 |
| 4608 | 52 | 72 | 21 | 65 | 74 | 22 |
| 18432 | 69 | 81 | 21 | 89 | 81 | 22 |
| 73728 | 91 | 86 | 20 | 131 | 85 | 22 |

Table 7: linear basis for $V_h$, cube

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 432 | 86 | 76 | 41 | 123 | 82 | 41 |
| 1728 | 113 | 93 | 63 | 168 | 103 | 72 |
| 6912 | 144 | 102 | 68 | 230 | 111 | 78 |
| 27648 | 184 | 121 | 81 | 313 | 132 | 93 |
| 110592 | 242 | 123 | 90 | 453 | 133 | 99 |

Table 8: linear basis for $V_h$, Fichera corner

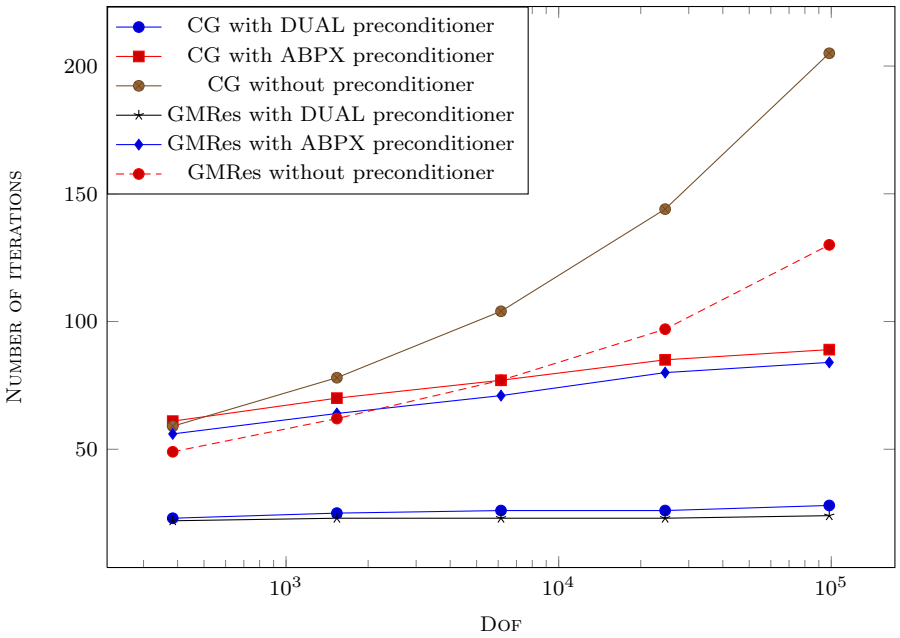



Figure 7: linear basis for $V_h$, sphere

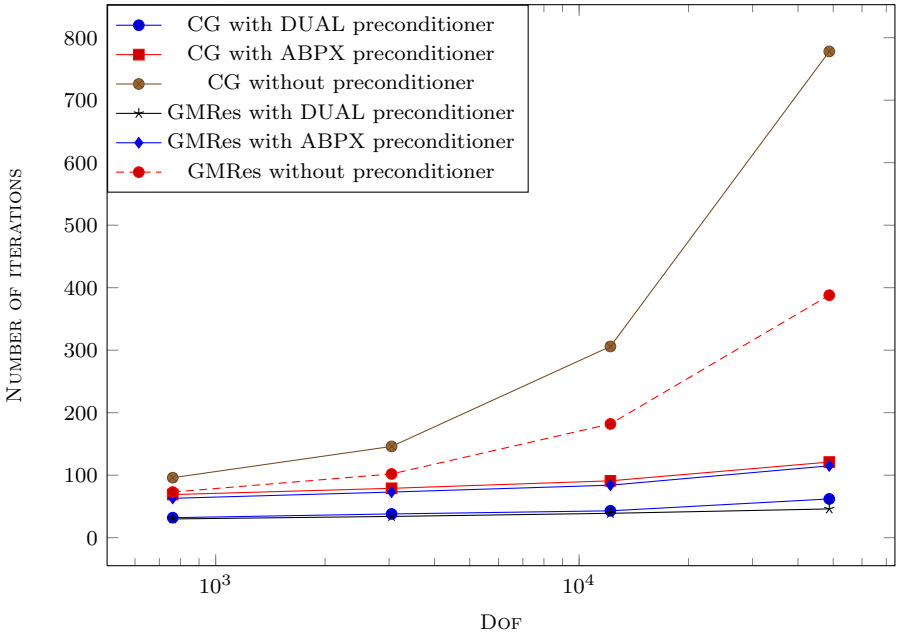



Figure 8: linear basis for $V_h$, cylinder

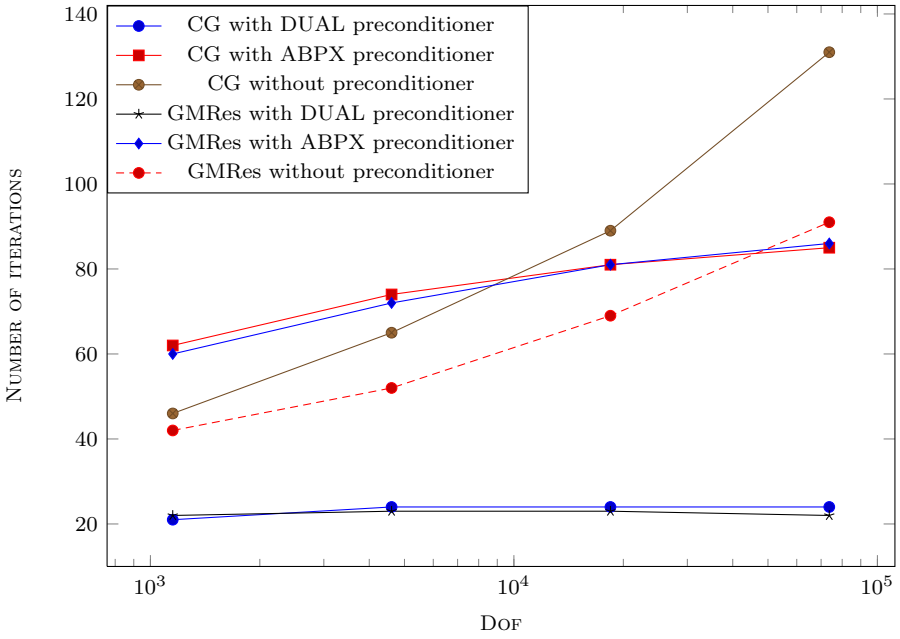



Figure 9: linear basis for $V_h$, cube

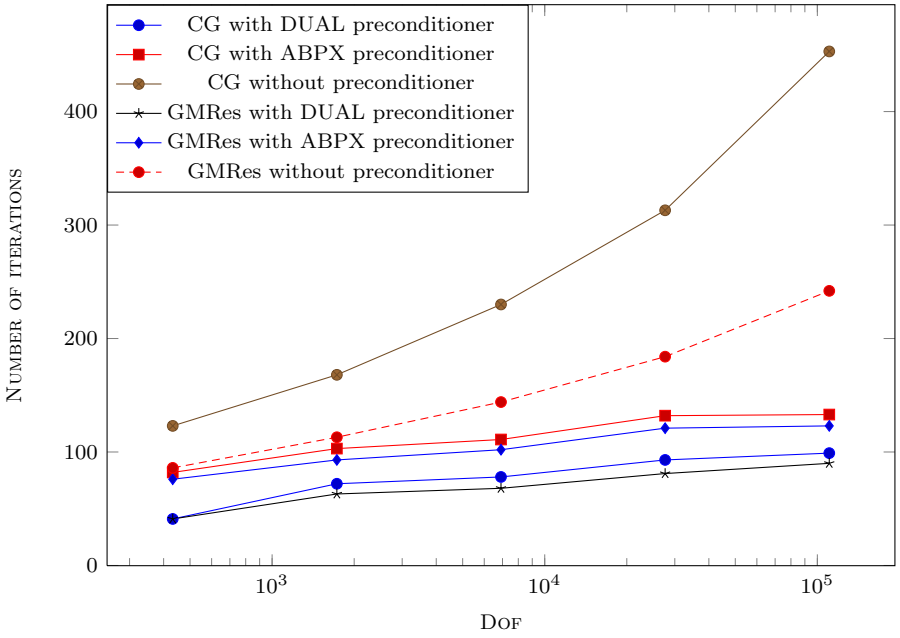



Figure 10: linear basis for $V_h$, Fichera corner

### 2.5.4 Test Models: Quadratic Basis

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 768 | 99 | 70 | 33 | 143 | 73 | 33 |
| 3072 | 129 | 80 | 35 | 195 | 85 | 37 |
| 12288 | 161 | 88 | 36 | 259 | 92 | 38 |
| 49152 | 196 | 97 | 36 | 345 | 100 | 40 |
| 196608 | 243 | 107 | 36 | 470 | 109 | 40 |

Table 9: quadratic basis for $V_h$, sphere

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 1524 | 149 | 78 | 42 | 234 | 82 | 42 |
| 6096 | 198 | 90 | 47 | 346 | 94 | 51 |
| 24384 | 350 | 103 | 56 | 769 | 106 | 63 |
| 97536 | 418 | 112 | 62 | 1278 | 121 | 73 |

Table 10: quadratic basis for $V_h$, cylinder

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 2304 | 80 | 75 | 30 | 108 | 74 | 29 |
| 9216 | 106 | 89 | 33 | 150 | 87 | 34 |
| 36864 | 142 | 98 | 33 | 220 | 95 | 35 |
| 147456 | 168 | 103 | 32 | 291 | 99 | 33 |

Table 11: quadratic basis for $V_h$, cube

| # of DOFs | GMRes w/o PC | GMRes ABPX PC | GMRes DUAL PC | CG w/o PC | CG ABPX PC | CG DUAL PC |
|---|---|---|---|---|---|---|
| 864 | 159 | 94 | 44 | 288 | 97 | 44 |
| 3456 | 229 | 116 | 45 | 412 | 122 | 49 |
| 13824 | 287 | 129 | 46 | 555 | 134 | 51 |
| 55296 | 398 | 150 | 48 | 829 | 157 | 55 |

Table 12: quadratic basis for $V_h$, Fichera corner

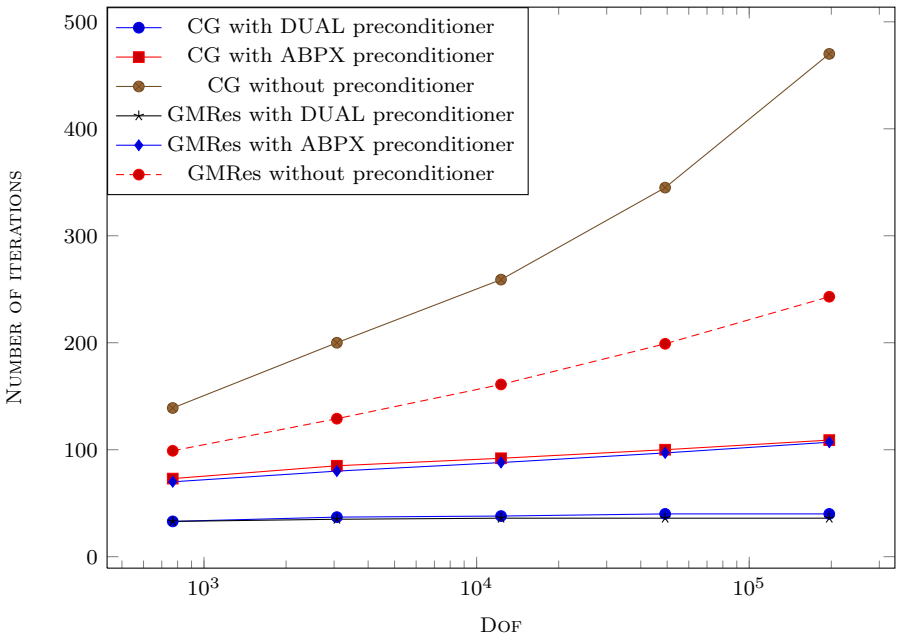



Figure 11: quadratic basis for $V_h$, sphere

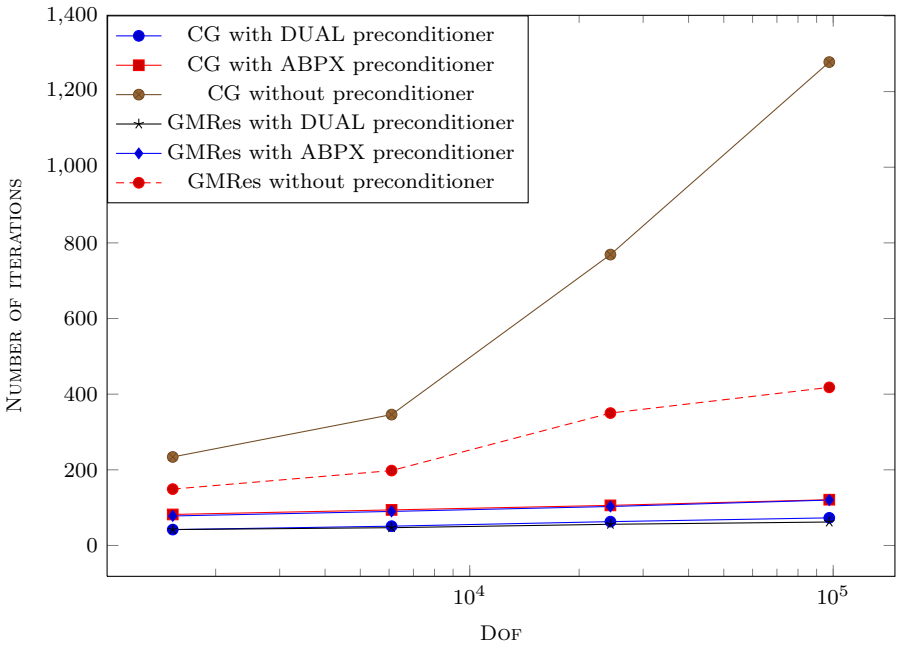



Figure 12: quadratic basis for $V_h$, cylinder

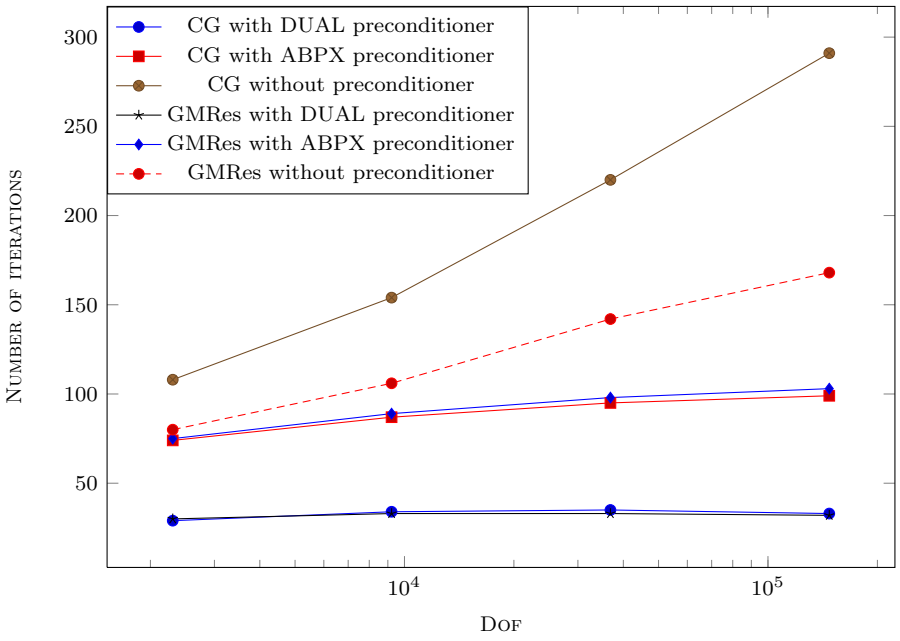



Figure 13: quadratic basis for $V_h$, cube

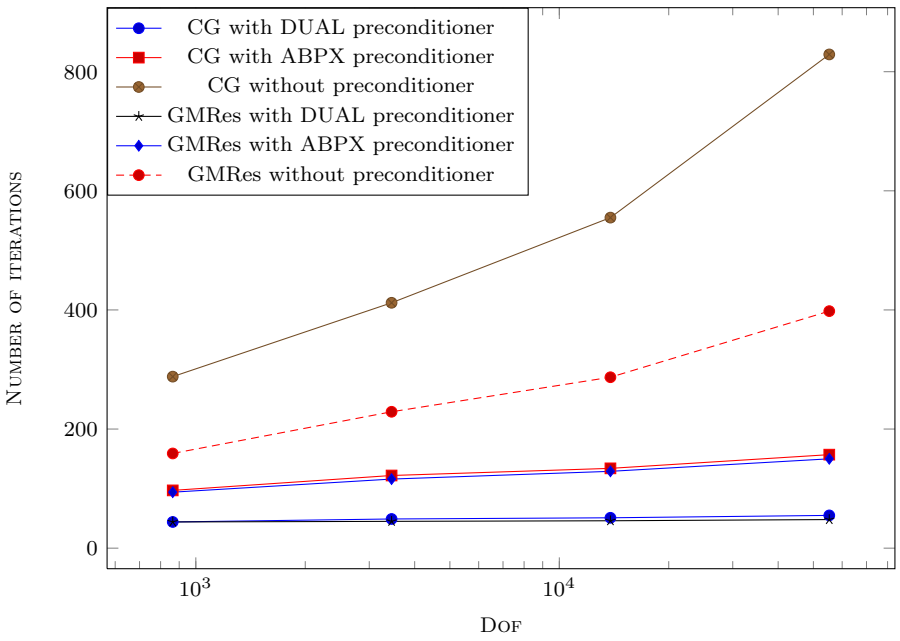



Figure 14: quadratic basis for $V_h$, Fichera corner

### 2.5.5 Industrial Application

The tests above validate our preconditioner. However it is interesting to evaluate the performance on a more sophisticated problem to make sure that our preconditioner does not somehow 'spoil' the solution.

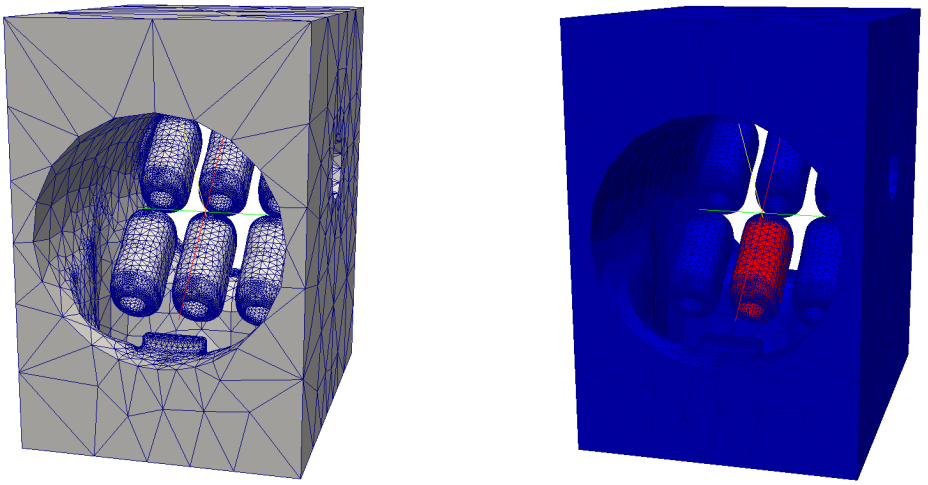
To this end we choose a model that consists of six electrodes in a casing. One of the electrodes has non-zero potential, i.e. $V = 1050$, the rest of the structure has free boundary (see Fig.15). And we are interested in the distribution of the charge density.

We thus solve the following equation with the constant Dirichlet data:

$$\begin{aligned} Vu &= Mu_D && \text{in } \Omega, \\ u_D &= g && \text{on } \partial\Omega, \\ g &= \begin{cases} 1050 & \text{on one electrode,} \\ 0 & \text{elsewhere.} \end{cases} \end{aligned} \tag{2.5.2}$$

For this model we only use dual Calderon preconditioner for the lowest order part.

Solution of the problem is represented on the Fig.16.



(a) Mesh (b) BC: Potential

Figure 15: Test model "six electrodes"

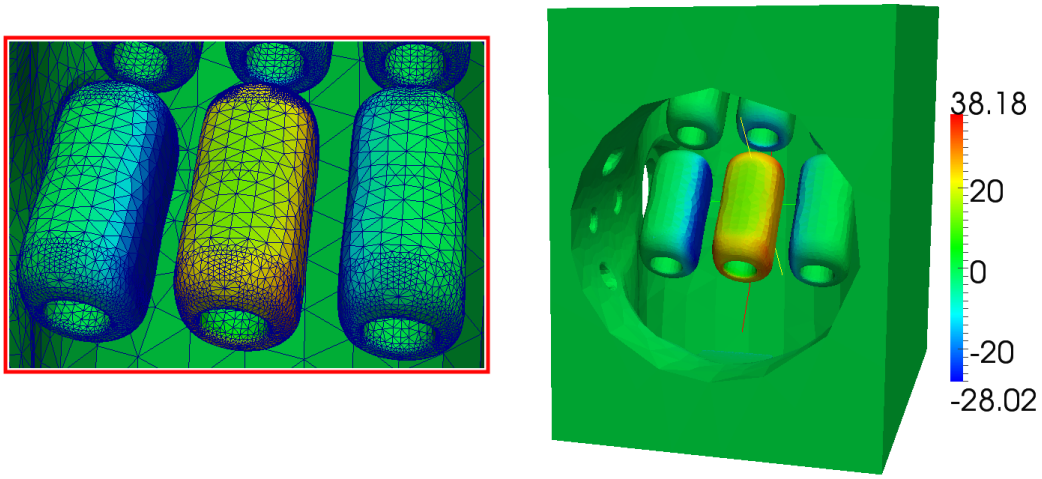



Figure 16: Test model "six electrodes": charge density distribution

Results with and without preconditioner correspond well: the qualitative pictures are identical, the difference in values is of the order $\approx 1e-6$.

| Basis | # of DOFs | GMRes w/o PC | GMRes DUAL PC | CG w/o PC | CG DUAL PC |
|---|---|---|---|---|---|
| CONSTANT | 50648 | 750 | 26 | 750 | 26 |
| LINEAR | 151944 | 976 | 47 | >10000 | 160 |
| QUADRATIC | 303888 | 1620 | 54 | >10000 | 165 |

Table 13: Test model "six electrodes": number of iterations

But this is a rather specific model: the mesh is very non-uniform (with $A_{max}/A_{min} \approx 1000$). This results in high conditional numbers of the system matrix, which is especially crucial for CG. We observe extremely high iteration numbers for pure CG. This makes the necessity of the preconditioner very obvious. And we can see that the preconditioned CG results in a moderate number of iterations as does also the preconditioned GMRes (see table 13). Partly it is because our hierarchical preconditioner is based on a Jacobi-like preconditioner, that balances the mesh size.

The overall calculation time for CG also decreased, making the total time of creating the preconditioner and solving the preconditioned system less than the time of solving the system straightforwardly (see table 14).

In case of GMRes time for solving the system without preconditioner is less than that with preconditioner. But the assembly of the preconditioner can be accelerated in *OpenMP*. Moreover if we make several calculation, i.e. have several load-cases for one model, using a preconditioner would be much more advantageous.

| Basis | # of DOFs | GMRes w/o PC | GMRes DUAL PC | CG w/o PC | CG DUAL PC | Assembling of PC |
|---|---|---|---|---|---|---|
| CONSTANT | 50648 | 291 | 253 | 292 | 254 | 1342 |
| LINEAR | 151944 | 1382 | 542 | $>10^4$ | 1474 | 1384 |
| QUADRATIC | 303888 | 5975 | 657 | $>10^4$ | 1950 | 1504 |

Table 14: Test model "six electrodes": calculation time (in seconds)

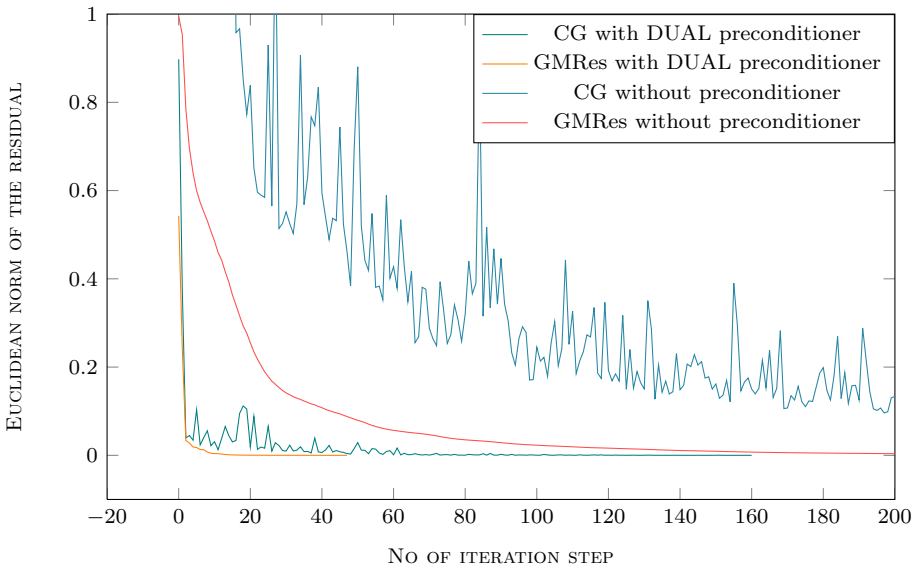



Figure 17: Test model "six electrodes": residuals for linear basis for $V_h$

# 3 Hierarchical Vector Bases

## 3.1 Quadratic Vector Functions

### 3.1.1 Definitions

On the reference triangle the $H(\mathrm{curl}_\Gamma)$- and $H(\mathrm{div}_\Gamma)$-conforming spaces are defined in the following way:

$$\begin{array}{l} R_k = (P_{k-1})^2 \oplus S_k, S_k = \{p \in (\tilde{P}_k)^2 : x \cdot p = 0\} \\ D_k = (P_{k-1})^2 \oplus (\tilde{P}_{k-1} \cdot x) \end{array} \tag{3.1.1}$$

where $\tilde{P}_k$ – are monomials of the order $k$. These spaces can be converted into each other by a 90 degree rotation of the vector fields.

With this definition we get that **curl** and **div** of the functions in $R_k$ and $D_k$ correspondingly are polynomials of the order $(k-1)$:

$$\begin{array}{l} \mathbf{curl} : R_k \to (P_{k-1})^2 \\ \mathbf{div} : D_k \to P_{k-1} \end{array} \tag{3.1.2}$$

In this paper we only work with flat triangles. On such elements both $R_k$ and $D_k$ have $3(k+1)$ edge functions and $2k$ face functions.

Thus the lowest (zero's) order $H(\mathrm{curl})$-conforming functions on a triangle are all only edge functions. In FEM case they are defined in [7] as:

$$\begin{array}{l} \mathbf{e^-_{12}} = \lambda_2 \mathbf{grad}\lambda_1 - \lambda_1 \mathbf{grad}\lambda_2 \\ \mathbf{e^-_{23}} = \lambda_3 \mathbf{grad}\lambda_2 - \lambda_2 \mathbf{grad}\lambda_3 \\ \mathbf{e^-_{31}} = \lambda_1 \mathbf{grad}\lambda_3 - \lambda_3 \mathbf{grad}\lambda_1 \end{array} \tag{3.1.3}$$

where $\lambda_i$ for $i = 1, 2, 3$ are barycentric coordinates on a triangle.

We base our BEM functions for $H(\mathrm{curl}_\Gamma)$-conforming basis on (3.1.3), only instead of the ordinary gradient **grad** we use surface gradient $\mathbf{grad}_\Gamma$:

$$\begin{array}{l} \mathbf{e^-_{12}} = \lambda_2 \mathbf{grad}_\Gamma\lambda_1 - \lambda_1 \mathbf{grad}_\Gamma\lambda_2 \\ \mathbf{e^-_{23}} = \lambda_3 \mathbf{grad}_\Gamma\lambda_2 - \lambda_2 \mathbf{grad}_\Gamma\lambda_3 \\ \mathbf{e^-_{31}} = \lambda_1 \mathbf{grad}_\Gamma\lambda_3 - \lambda_3 \mathbf{grad}_\Gamma\lambda_1 \end{array} \tag{3.1.4}$$

where $\lambda_i$ for $i = 1, 2, 3$ are barycentric coordinates on a triangle.

REMARK: in the case of triangular mesh we work not only on a given triangle, but also on the reference triangle. Thus we introduce notations: for any entity $a$ we denote by plain $a$ the value on a given triangle of the mesh and by $\hat{a}$ on the reference triangle.

The lowest order $H(\mathrm{div}_\Gamma)$-conforming functions on a triangle can be obtained from $H(\mathrm{curl}_\Gamma)$ functions by simply rotating the latter, i.e.

$$\mathbf{e^-_{ij}}_{div} = \mathbf{e^-_{ij}}_{curl} \times \mathbf{n}, i = 1, 2, 3, \tag{3.1.5}$$

where $\mathbf{n}$ is the outer normal vector.

The div-functions can also be rewritten in a more easily geometrically interpreted way:

$$\mathbf{e_{ij}^-} = \frac{\mathbf{x} - \mathbf{x}_k}{2A} \;\; i,j,k = 1,2,3, \tag{3.1.6}$$

where $\overline{x}_i, i = 1,2,3$ – are the vertices of the triangle and $A$ – is its area.

These linear functions have already been incorporated into BETL. We intend to implement the next spaces – the first order $H(\mathrm{curl}_\Gamma)$- and $H(\mathrm{div}_\Gamma)$-conforming functions.

The functions of $R_1$ consist of the functions of $R_0$ (3.1.4), which are supplemented with 3 more edge functions:

$$\begin{array}{l} \mathbf{e_{12}^+} = \lambda_2 \mathbf{grad}_\Gamma \lambda_1 + \lambda_1 \mathbf{grad}_\Gamma \lambda_2 \\ \mathbf{e_{23}^+} = \lambda_3 \mathbf{grad}_\Gamma \lambda_2 + \lambda_2 \mathbf{grad}_\Gamma \lambda_3 \\ \mathbf{e_{31}^+} = \lambda_1 \mathbf{grad}_\Gamma \lambda_3 + \lambda_3 \mathbf{grad}_\Gamma \lambda_1 \end{array} \tag{3.1.7}$$

and 2 face functions:

$$\begin{array}{l} \mathbf{f_1} = \lambda_2 \mathbf{e_{31}^-} \\ \mathbf{f_2} = \lambda_3 \mathbf{e_{12}^-} \end{array} \tag{3.1.8}$$

As in the zero's order case, the corresponding $D_1$ functions can be obtained from $R_1$ functions by simple rotation, i.e. by cross-product with the normal vector.

### 3.1.2 Quadratic Vector Functions on the Reference Triangle

As in scalar case we first want to evaluate our basis functions on the reference triangle

$$\hat{T} := \{(\hat{x}_1, \hat{x}_2) : 0 < \hat{x}_2 < \hat{x}_1 < 1\}. \tag{3.1.9}$$

For this purpose we need to make some definitions.

The mapping from $\hat{T}$ to an arbitrary triangle $T$ is given by:

$$x(\hat{x}) = \sum_{i=1}^{3} \mathbf{x}_i \phi_i(\hat{x}), \quad \hat{x} \in \mathbb{R}^2, x \in \mathbb{R}^3, \tag{3.1.10}$$

where $\phi_i(\hat{x})$ – are interpolatory nodal basis functions on $\hat{T}$ and $\mathbf{x}_i$ – are the vertices of the triangle $T$, $i = 1,2,3$.

Then if we define tangent vectors as

$$\mathbf{t}_i := \frac{\partial x}{\partial \hat{x}_i}, \tag{3.1.11}$$

we can write the Jacobi matrix of this mapping as

$$\mathbf{J} := \begin{bmatrix} \mathbf{t}_1 & \mathbf{t}_2 \end{bmatrix} \tag{3.1.12}$$

Subsequently we introduce the Gramian matrix and its determinant:

$$\mathbf{G} := \mathbf{J}^T\mathbf{J}, \quad g = \det(\mathbf{G}). \tag{3.1.13}$$

Finally for Dirichlet trace we have the following transformation rule:

$$\gamma_D\mathbf{u}(\mathbf{x}) = \mathbf{n} \times (\mathbf{u}(\mathbf{x}) \times \mathbf{n}) = \mathbf{J}\mathbf{G}^{-1}\hat{\mathbf{u}}(\hat{\mathbf{x}}) \ , \tag{3.1.14}$$

where $\hat{\mathbf{u}}(\hat{\mathbf{x}}) := \mathbf{J}^\top\mathbf{u}(\mathbf{x}) = \mathbf{J}^\top\gamma_D\mathbf{u}(\mathbf{x})$ is the trace of the function $\mathbf{u}(\mathbf{x})$ on the reference triangle, $\mathbf{x} \in \mathbb{R}^3$ and $\hat{\mathbf{x}} \in \mathbb{R}^2$.

Now in order to calculate the values of the quadratic vector functions, we need to evaluate $\mathbf{grad}_\Gamma\lambda_i$. For the surface gradient of any scalar function $\varphi$ we get

$$\mathbf{grad}_\Gamma\varphi := \gamma_D\mathbf{grad}\varphi = \mathbf{J}\mathbf{G}^{-1}\ \widehat{\mathbf{grad}}\varphi, \tag{3.1.15}$$

where $\widehat{\mathbf{grad}}\varphi$ is the ordinary $2D$ gradient on the reference triangle:

$$\widehat{\mathbf{grad}}\varphi := \begin{bmatrix} \partial\varphi/\partial\hat{x}_1 \\ \partial\varphi/\partial\hat{x}_2 \end{bmatrix} \tag{3.1.16}$$

With

$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} 1-\hat{x}_1 \\ \hat{x}_1 - \hat{x}_2 \\ \hat{x}_2 \end{bmatrix} \tag{3.1.17}$$

the gradients $\widehat{\mathbf{grad}}\lambda_i$ are

$$\widehat{\mathbf{grad}}\lambda_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}, \quad \widehat{\mathbf{grad}}\lambda_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \widehat{\mathbf{grad}}\lambda_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \ . \tag{3.1.18}$$

Hence, the local edge- and face-functions are

$$\begin{gathered} \hat{\mathbf{e}}_{12}^- = \begin{bmatrix} \lambda_1+\lambda_2 \\ -\lambda_1 \end{bmatrix}, \quad \hat{\mathbf{e}}_{23}^- = \begin{bmatrix} -\lambda_3 \\ \lambda_2+\lambda_3 \end{bmatrix}, \quad \hat{\mathbf{e}}_{31}^- = \begin{bmatrix} -\lambda_3 \\ -\lambda_1 \end{bmatrix} \\ \hat{\mathbf{e}}_{12}^+ = \begin{bmatrix} \lambda_1-\lambda_2 \\ -\lambda_1 \end{bmatrix}, \quad \hat{\mathbf{e}}_{23}^+ = \begin{bmatrix} \lambda_3 \\ \lambda_2-\lambda_3 \end{bmatrix}, \quad \hat{\mathbf{e}}_{31}^+ = \begin{bmatrix} -\lambda_3 \\ \lambda_1 \end{bmatrix} \\ \hat{\mathbf{f}}_1 = \lambda_3 \begin{bmatrix} \lambda_1+\lambda_2 \\ -\lambda_1 \end{bmatrix}, \quad \hat{\mathbf{f}}_2 = \lambda_1 \begin{bmatrix} -\lambda_3 \\ \lambda_2+\lambda_3 \end{bmatrix} \ . \end{gathered} \tag{3.1.19}$$

### 3.1.3 Quadratic Vector Functions on an Arbitrary Triangle

Following (3.1.15) we get that edge- and face functions on any given triangle can be obtained from the functions on the reference triangle by multiplication by $\mathbf{JG}^{-1}$. However we also have to account for the orientation of the edges in the triangle.

The orientation in BETL is induced by vertices numeration. An arbitrary edge $E_{ab}$ between nodes with indices $a$ and $b$ is positively oriented, if $b>a$. Then we say that $\sigma_{E_{ij}} = 1$, where $\sigma$ stands for orientation. Otherwise $\sigma_{E_{ij}} = -1$.

A flat triangle is stored as a triplet of vertex indices, i.e. $T = (i, j, k)$. This triangle has three edges: $E_{ij}$, $E_{jk}$ and $E_{ki}$. They correspond to edges $\hat{E}_{01}$, $\hat{E}_{12}$ and $\hat{E}_{20}$ of the reference triangle $\hat{T}$ correspondingly. In (3.1.19) we assume that on $\hat{T}$ all edges are positively oriented. Thus if the orientation of the edges of the triangle $T$ differ from those of their prototypes on $\hat{T}$, we have to correct them.

Hence for $\mathbf{e}_{ij}^-$ and for $\mathbf{f}_i$ we add an orientation factor $\sigma_{E_{ij}}$. Functions $\mathbf{e}_{ij}^+$ are symmetric and do not require orientation correction:

$$\begin{array}{lcll} \mathbf{e}_{ij}^- & = & \sigma_{E_{ij}} & \mathbf{JG}^{-1}\hat{\mathbf{e}}_{ij}^- \\ \mathbf{e}_{ij}^+ & = & & \mathbf{JG}^{-1}\hat{\mathbf{e}}_{ij}^+ \\ \mathbf{f}_1 & = & \sigma_{E_{31}} & \mathbf{JG}^{-1}\hat{\mathbf{f}}_1 \\ \mathbf{f}_2 & = & \sigma_{E_{12}} & \mathbf{JG}^{-1}\hat{\mathbf{f}}_2 \end{array} \tag{3.1.20}$$

### 3.1.4 $\mathbf{curl}_\Gamma$ and $\mathbf{div}_\Gamma$ of the Quadratic Vector Functions

For the vector surface curl we have

$$\mathbf{curl}_\Gamma\varphi := \mathbf{grad}\varphi \times \mathbf{n} = \frac{1}{\sqrt{g}}\mathbf{J}\mathbf{H}\widehat{\mathbf{grad}}\varphi, \quad \mathbf{H} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}. \tag{3.1.21}$$

Matrix $\mathbf{H}$ preforms a 90° clockwise rotation in 2D plane.

This is equivalent to

$$\mathbf{curl}_\Gamma\varphi = \frac{1}{\sqrt{g}}\mathbf{J}\widehat{\mathbf{curl}}\varphi, \tag{3.1.22}$$

where $\widehat{\mathbf{curl}}$ is given by

$$\widehat{\mathbf{curl}}\varphi := \begin{bmatrix} \partial\varphi/\partial\hat{x}_2 \\ -\partial\varphi/\partial\hat{x}_1 \end{bmatrix} \tag{3.1.23}$$

Thus the scalar surface curl of vector valued functions can be expressed via the 2-dimensional scalar curl in the reference domain in the following way

$$\mathrm{curl}_\Gamma \mathbf{u} = \frac{1}{\sqrt{g}} \mathbf{J}^T \widehat{\mathrm{curl}} \hat{\mathbf{u}} \tag{3.1.24}$$

where $\widehat{\mathrm{curl}}$ is given by

$$\widehat{\mathrm{curl}} \hat{\mathbf{u}} := \frac{\partial \hat{u}_2(\hat{x})}{\partial \hat{x}_1} - \frac{\partial \hat{u}_1(\hat{x})}{\partial \hat{x}_2} \ . \tag{3.1.25}$$

As in (3.1.20) we have to make a correction for orientation of the edges on the original triangle, i.e.

$$\begin{array}{rcl} \mathrm{curl}_\Gamma \mathbf{e}_{ij}^- & = & \frac{1}{\sqrt{g}} \sigma_{E_{ij}} \mathbf{J}^T \widehat{\mathrm{curl}} \hat{\mathbf{e}}_{ij}^-, \\ \mathrm{curl}_\Gamma \mathbf{e}_{ij}^+ & = & \frac{1}{\sqrt{g}} \mathbf{J}^T \widehat{\mathrm{curl}} \hat{\mathbf{e}}_{ij}^+ \\ \mathrm{curl}_\Gamma \mathbf{f}_1 & = & \frac{1}{\sqrt{g}} \sigma_{E_{31}} \mathbf{J}^T \widehat{\mathrm{curl}} \hat{\mathbf{f}}_1 \\ \mathrm{curl}_\Gamma \mathbf{f}_2 & = & \frac{1}{\sqrt{g}} \sigma_{E_{12}} \mathbf{J}^T \widehat{\mathrm{curl}} \hat{\mathbf{f}}_2 \end{array} \tag{3.1.26}$$

On the reference triangle we obtain:

$$\begin{aligned} \widehat{\mathrm{curl}} \hat{\mathbf{e}}_{ij}^- &= -2 \\ \widehat{\mathrm{curl}} \hat{\mathbf{e}}_{ij}^+ &= 0 \\ \widehat{\mathrm{curl}} \hat{\mathbf{f}}_1 &= 3\lambda_2 - 1 \\ \widehat{\mathrm{curl}} \hat{\mathbf{f}}_2 &= 3\lambda_3 - 1 \end{aligned} \tag{3.1.27}$$

Values of surface divergence of $H(\mathrm{div}_\Gamma)$-conforming functions and scalar surface curl of $H(\mathrm{curl}_\Gamma)$-conforming functions coincide, and thus from (2.4.3), (3.1.26) and (3.1.27) we get:

$$\begin{bmatrix} \mathrm{div}_\Gamma \mathbf{e}_{12_{div}}^- \\ \mathrm{div}_\Gamma \mathbf{e}_{23_{div}}^- \\ \mathrm{div}_\Gamma \mathbf{e}_{31_{div}}^- \\ \mathrm{div}_\Gamma \mathbf{e}_{12_{div}}^+ \\ \mathrm{div}_\Gamma \mathbf{e}_{32_{div}}^+ \\ \mathrm{div}_\Gamma \mathbf{e}_{31_{div}}^+ \\ \mathrm{div}_\Gamma \mathbf{f}_{1_{div}} \\ \mathrm{div}_\Gamma \mathbf{f}_{2_{div}} \end{bmatrix} = \begin{bmatrix} \mathrm{curl}_\Gamma \mathbf{e}_{12_{curl}}^- \\ \mathrm{curl}_\Gamma \mathbf{e}_{23_{curl}}^- \\ \mathrm{curl}_\Gamma \mathbf{e}_{31_{curl}}^- \\ \mathrm{curl}_\Gamma \mathbf{e}_{12_{curl}}^+ \\ \mathrm{curl}_\Gamma \mathbf{e}_{32_{curl}}^+ \\ \mathrm{curl}_\Gamma \mathbf{e}_{31_{curl}}^+ \\ \mathrm{curl}_\Gamma \mathbf{f}_{1_{curl}} \\ \mathrm{curl}_\Gamma \mathbf{f}_{2_{curl}} \end{bmatrix} = \frac{1}{\sqrt{g}} \underbrace{\begin{bmatrix} -2\sigma_{E_{12}} & -2\sigma_{E_{12}} & -2\sigma_{E_{12}} \\ -2\sigma_{E_{23}} & -2\sigma_{E_{23}} & -2\sigma_{E_{23}} \\ -2\sigma_{E_{31}} & -2\sigma_{E_{31}} & -2\sigma_{E_{31}} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ -\sigma_{E_{31}} & 2\sigma_{E_{31}} & -\sigma_{E_{31}} \\ -\sigma_{E_{12}} & -\sigma_{E_{12}} & 2\sigma_{E_{12}} \end{bmatrix}}_{D_{loc}} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} \tag{3.1.28}$$

### 3.2 Maxwell Matrix for Quadratic Basis

The general Maxwell problem for electric field reads as follows:

$$\begin{cases} \mathbf{curlcurlE} + \kappa^2 \mathbf{E} = 0 & \text{in } \Omega \\ \mathbf{E} = \mathbf{E}^{inc} & \text{in } \Gamma = \partial\Omega \\ \text{Silver-Müller radiation condition} & \text{at } \infty \end{cases} \tag{3.2.1}$$

where $\mathbf{E}$ – is the unknown electric field, $\mathbf{E}^{inc}$ – is the incoming electric field and $\kappa = \omega\sqrt{\epsilon\mu} \in \mathbb{C}$ is the wave number associated to the frequency $\omega$.

Using the Stratton-Chu representation formula we arrive at the following variational boundary integral equation of the first kind for the unknown tangential component of the electric field $\mathbf{u}$ [2, 5]:

$$\begin{array}{l} \text{Find } \mathbf{u} \in H^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \text{ such that} \\ \Psi(\mathbf{u}, \mathbf{v}) - \frac{1}{\kappa^2}\Phi(\mathrm{div}_\Gamma \mathbf{u}, \mathrm{div}_\Gamma \mathbf{v}) = (\mathbf{f}, \mathbf{v}), \quad \mathbf{f} \in H^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \\ \text{is fulfilled } \forall \mathbf{v} \in H^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \end{array} \tag{3.2.2}$$

where the sesqui-linear forms $\Phi(\cdot,\cdot)$ and $\Psi(\cdot,\cdot)$ are scalar and vectorial Helmholtz single layer potentials:

$$\Phi(u, v) := \int_\Gamma \int_\Gamma G(\mathbf{y} - \mathbf{x}) u(\mathbf{x}) v(\mathbf{y}) ds(\mathbf{y}) ds(\mathbf{x}), \tag{3.2.3a}$$

$$\Psi(\mathbf{u}, \mathbf{v}) := \int_\Gamma \int_\Gamma G(\mathbf{y} - \mathbf{x}) \left(\mathbf{u}(\mathbf{x}), \mathbf{v}(\mathbf{y})\right) ds(\mathbf{y}) ds(\mathbf{x}). \tag{3.2.3b}$$

based on the fundamental solution of the Helmholtz equation

$$G(\mathbf{z}) = \frac{\exp(i\kappa|\mathbf{z}|)}{|\mathbf{z}|}, \quad \mathbf{z} \neq \mathbf{0}. \tag{3.2.4}$$

We define the sesqui-linear form corresponding to the left hand side of the equation (3.2.2) as $V(\cdot,\cdot)$:

$$V(\mathbf{u}, \mathbf{v}) := \Psi(\mathbf{u}, \mathbf{v}) - \frac{1}{\kappa^2}\Phi(\mathrm{div}_\Gamma \mathbf{u}, \mathrm{div}_\Gamma \mathbf{v}). \tag{3.2.5}$$

This is the Maxwell counterpart of the single layer potential operator.

A Galerkin discretization of this boundary integral equation by means of the vector functions described in Sec.3 results in a system of linear equations:

$$V_h \mathbf{u}_h = \mathbf{b}_h, \tag{3.2.6}$$

where $\mathbf{u}_h$ is the vector of unknowns and $\mathbf{b}_h$ is the right hand side vector.

The matrix $V_h$ is a composite matrix, based on two similar BEM matrices. However in BETL we cannot directly pass divergences of the basis functions into the BEM matrix class. Thus we propose the following approach for evaluating $V_h$.

In case of linear basis functions divergences are constant (3.1.28):

$$\mathrm{div}\mathbf{e}_{ij}^{-} = -\frac{2}{\sqrt{g}}\sigma_{E_{ij}} \tag{3.2.7}$$

This allows us to assemble the $V_h$ matrix in the following way:

$$V_h = \Psi_h - \frac{1}{\kappa^2} D_h \; \Phi_h \; D_h^T, \tag{3.2.8}$$

where $\Psi_h$ – is the Galerkin matrix of the vectorial Helmholtz single layer potential (3.2.3b) evaluated on the linear vector basis functions $\mathbf{e}_{ij}^{-}$, $\Phi_h$ – is the Galerkin matrix of the scalar Helmholtz single layer potential (3.2.3a) evaluated on the constant scalar basis functions and $D_h$ – is the divergence matrix constructed from $3 \times 1$ blocks $D_{loc}$ corresponding to each triangle.

$$D_{loc} = \begin{pmatrix} -\frac{2}{\sqrt{g}}\sigma_{E_{12}} \\ -\frac{2}{\sqrt{g}}\sigma_{E_{23}} \\ -\frac{2}{\sqrt{g}}\sigma_{E_{31}} \end{pmatrix} \tag{3.2.9}$$

We use the same structure (3.2.8) for quadratic vector basis, only now $\Psi_h$ – is the Galerkin matrix of the vectorial Helmholtz single layer potential evaluated on the quadratic vector basis functions, $\Phi_h$ – is the Galerkin matrix of the scalar Helmholtz single layer potential evaluated on the linear nodal basis functions and $D_h$ – is the divergence matrix constructed from $8 \times 3$ blocks $D_{loc}$ from (3.1.28).

### 3.3 Local Interpolation Routines

The new quadratic functions should support the interpolation routine as well. I.e. for a given vector function $\mathbf{u}$ we have to be able to approximate its Dirichlet trace on any triangle in the following way:

$$\begin{aligned} \gamma_D \mathbf{u} \approx \; & \alpha_{12}^{-}\mathbf{e}_{12}^{-} + \alpha_{23}^{-}\mathbf{e}_{23}^{-} + \alpha_{31}^{-}\mathbf{e}_{31}^{-} + \\ & \alpha_{12}^{+}\mathbf{e}_{12}^{+} + \alpha_{23}^{+}\mathbf{e}_{23}^{+} + \alpha_{31}^{+}\mathbf{e}_{31}^{+} + \\ & \beta_1\mathbf{f}_1 + \beta_2\mathbf{f}_2 \end{aligned} \tag{3.3.1}$$

Naturally two slightly different strategies for edge and face functions are used.

### 3.3.1 Edge functions

For the edge $E_{ij}$ between nodes $i$ and $j$ of a triangle $T$ we define the interpolation coefficients corresponding to edge basis functions in the following integral form:

$$\alpha_{ij}^{\sigma} = \int_{E_{ij}} m^{\sigma}(\gamma_D \mathbf{u})^{\top} \mathbf{t}_{ij} dE = \int_0^1 \hat{m}^{\sigma}(\hat{\mathbf{u}}, \hat{\mathbf{t}}_{ij}) ds \tag{3.3.2}$$

where $\mathbf{t}_{ij}$ denotes the respective normalized tangent vector, $\sigma = \{-, +\}$ and $m^{\sigma}$ – are some moments yet to be determined. Hat notations as usual correspond to the same entities on the reference triangle.

We choose moments $\hat{m}^{+}$ and $\hat{m}^{-}$ such that

$$\begin{array}{l} \int_0^1 m^{\sigma_1} \left(\hat{\mathbf{t}}_{ij}, \hat{\mathbf{e}}_{kl}^{\sigma_2}\right) \, ds = \delta_{(ij)(kl)} \delta_{\sigma_1 \sigma_2} \\ \int_0^1 m^{\sigma} \left(\hat{\mathbf{t}}_{ij}, \hat{\mathbf{f}}_l\right) \, ds = 0 \end{array} \tag{3.3.3}$$

This gives us

$$m^{-}(s) := 1, \quad m^{+}(s) := 3(1 - 2s). \tag{3.3.4}$$

### 3.3.2 Face functions

Similarly to the edge case for face functions we want the coefficients to be defined as

$$\begin{array}{l} \beta_i = \int_T \mathbf{u}^{\top} \, (\mathbf{n} \times \widetilde{\mathbf{m}}_i) \, dT = \int_{\hat{T}} \left(\widehat{\mathbf{m}}, \mathbf{J}^{\top} \mathbf{u}\right) \, d\hat{T} \,, \\ \text{where} \quad \widehat{\mathbf{m}} := -\mathbf{H} \widehat{\widetilde{\mathbf{m}}} \,, \mathbf{H} = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \end{array} \tag{3.3.5}$$

In particular we choose the moments $\widehat{\mathbf{m}}_1$, $\widehat{\mathbf{m}}_2$ such that

$$\int_{\hat{T}} \left(\widehat{\mathbf{m}}_i, \hat{\mathbf{f}}_j\right) \, d\hat{T} = \delta_{ij} \,. \tag{3.3.6}$$

This yields

$$\widehat{\mathbf{m}}_1 = \begin{bmatrix} 16 \\ 8 \end{bmatrix} \,, \quad \widehat{\mathbf{m}}_2 = \begin{bmatrix} 8 \\ 16 \end{bmatrix} \,. \tag{3.3.7}$$

In this case however

$$\int_{\hat{T}} \left(\widehat{\mathbf{m}}_i, \hat{\mathbf{e}}_{jk}^{\sigma}\right) \, d\hat{T} \neq 0 \,. \tag{3.3.8}$$

### 3.3.3 Final Interpolation Scheme for Triangles

Finally from Sec.3.3.2 and Sec.3.3.1 we have

$$\underbrace{\left[\begin{array}{ccc|ccc|ccc} 1 & & & & & & & & \\ & 1 & & & & & & & \\ & & 1 & & & & & & \\ \hline & & & 1 & & & & & \\ & & & & 1 & & & & \\ & & & & & 1 & & & \\ \hline 4 & & -4 & -\frac{4}{3} & \frac{8}{3} & -\frac{4}{3} & 1 & & \\ & 4 & -4 & -\frac{8}{3} & \frac{4}{3} & \frac{4}{3} & & 1 & \end{array}\right]}_{\mathbf{C}} \underbrace{\left[\begin{array}{c} \alpha_{12}^{-} \\ \alpha_{23}^{-} \\ \alpha_{31}^{-} \\ \alpha_{12}^{+} \\ \alpha_{23}^{+} \\ \alpha_{31}^{+} \\ \beta_1 \\ \beta_2 \end{array}\right]}_{\alpha} = \underbrace{\left[\begin{array}{c} \int_0^1 (\hat{\mathbf{t}}_{12}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_0^1 (\hat{\mathbf{t}}_{23}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_0^1 (\hat{\mathbf{t}}_{31}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_0^1 3(1-2s)(\hat{\mathbf{t}}_{12}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_0^1 3(1-2s)(\hat{\mathbf{t}}_{23}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_0^1 3(1-2s)(\hat{\mathbf{t}}_{31}, \mathbf{J}^\top \mathbf{u})\, ds \\ \int_{\hat{T}} (\hat{\mathbf{m}}_1, \mathbf{J}^\top \mathbf{u})\, d\hat{T} \\ \int_{\hat{T}} (\hat{\mathbf{m}}_2, \mathbf{J}^\top \mathbf{u})\, d\hat{T} \end{array}\right]}_{\hat{\alpha}}. \tag{3.3.9}$$

Thus for the resulting interpolation coefficients $\alpha$ we get

$$\alpha = \mathbf{C}^{-1}\hat{\alpha}. \tag{3.3.10}$$

## 3.4 Interpolation Errors for $R_0$ and $R_1$ Spaces

To validate the implementation of the interpolation we calculate an interpolation error using the interpolation scheme described in Sec.3.3.3:

$$\|\mathbf{e}\|_{L_2} = \|\mathbf{u} - \mathbf{u}_h\|_{L_2}, \tag{3.4.1}$$

where $\mathbf{u}$ – is a plane wave function (4.3.1), $\mathbf{u}_h$ – is $\mathbf{u}$ interpolated using linear or quadratic vector basis functions.

| $l$ | $h$ | $e_{R_0}$ | $\frac{e_{R_0}^{l-1}}{e_{R_0}^{l}}$ | $e_{R_1}$ | $\frac{e_{R_1}^{l-1}}{e_{R_1}^{l}}$ |
|---|---|---|---|---|---|
| 0 | 0.1506 | 0.0792 | - | 0.0054 | - |
| 1 | 0.0773 | 0.0411 | 1.9277 | 0.0015 | 3.5905 |
| 2 | 0.0389 | 0.0208 | 1.9793 | 0.0004 | 3.8688 |
| 3 | 0.0195 | 0.0104 | 1.9943 | 9.9e-05 | 3.9544 |
| 4 | 0.0098 | 0.0052 | 2.0000 | 2.5e-05 | 3.9858 |
| 5 | 0.0049 | 0.0026 | 2.0000 | 6.2e-06 | 3.9961 |
| 6 | 0.0024 | 0.0013 | 2.0000 | 1.5e-06 | 3.9974 |

Table 15: Interpolation error for linear and quadratic vector functions

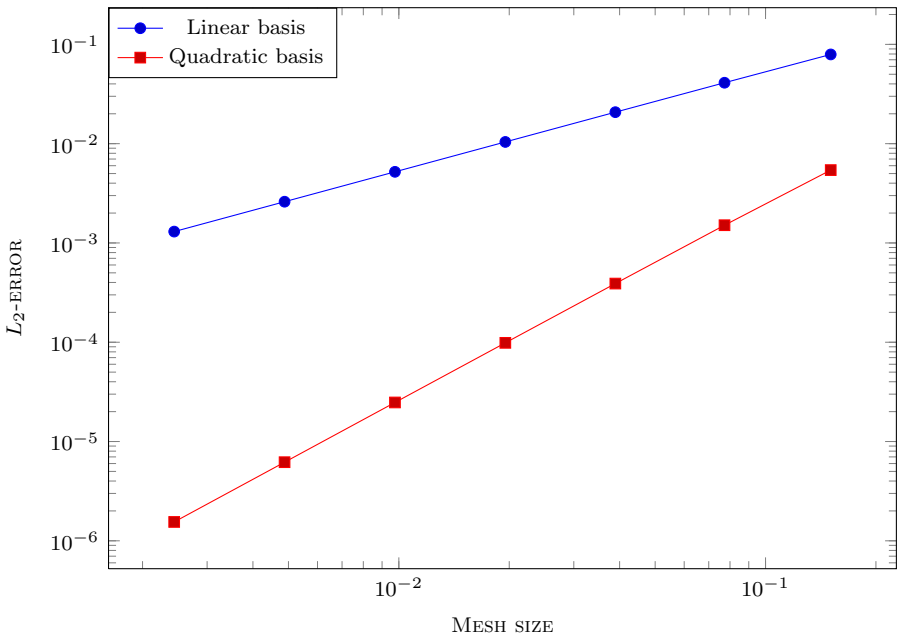



Figure 18: Interpolation error for linear and quadratic vector functions

From 15 we can clearly see that as expected

$$\begin{aligned} \|\mathbf{e}\|_{L_2}^{R_0} &\leq Ch, \\ \|\mathbf{e}\|_{L_2}^{R_1} &\leq Ch^2. \end{aligned} \tag{3.4.2}$$

# 4 Edge Calderon Preconditioner

## 4.1 Preconditioner Formula

We construct the preconditioner following the article [1].

Imagine we have two meshes: original and barycentric refinement. And we have 4 bases associated with these meshes:

$\{\Phi_i\}_{i=1}^n$ – linear, continuous, div-conforming basis functions on the original mesh;

$\{\varphi_i^d\}_{i=1}^m$ – linear, continuous, div-conforming basis functions on the barycentric refinement;

$\{\varphi_i^c\}_{i=1}^m$ – linear, continuous, curl-conforming basis functions on the barycentric refinement;

$\{\psi_i\}_{i=1}^n$ – linear, continuous, div- and quasicurl-conforming basis functions on the dual mesh defined in [1].

The latter functions are strictly div-conforming by construction. They also are quasicurl-conforming in the sense that the Gram matrix linking $\{\psi_i\}_{i=1}^n$ and $\{\Phi_i\}_{i=1}^n$ bases is well conditioned.

Imagine we have some bilinear form $Z(\cdot,\cdot)$ that we want to precondition. And the corresponding system is:

$$Z_h x_h = b_h \tag{4.1.1}$$

where $Z_h$ – is the matrix evaluated on $\{\Phi_i\}_{i=1}^n$.

We want to use similar matrix as a preconditioner, i.e. $Z_{h,prec}$ on $\{\psi_i\}_{i=1}^n$.

Instead of calculating two BEM matrices we only calculate one: $Z_{h,b}$ – similar matrix on the barycentric refinement, i.e. on $\{\varphi_i^d\}_{i=1}^m$.

Afterwards we create two incidence matrices $R$ and $P$ between barycentric refinement and original or dual mesh correspondingly.

Then we can get both $Z_h$ and $Z_{h,prec}$ from $Z_{h,b}$:

$$\begin{array}{ll} Z_h & = R^T Z_{h,b} R, \\ Z_{h,prec} & = P^T Z_{h,b} P \end{array} \tag{4.1.2}$$

Thus we come to the system

$$Z_{h,prec} G^{-1} Z_h x_h = \hat{b}_h = Z_{h,prec} G^{-1} b_h. \tag{4.1.3}$$

In (4.1.3) $G$ is the mass matrix between $\{\Phi_i\}_{i=1}^n$ and $\{\psi_i\}_{i=1}^n$ and it has the following form

$$G = R^T G_b P, \tag{4.1.4}$$

where $G_b$ is the mass matrix between linear, continuous, div- and curl- conforming basis functions on the barycentric refinement, i.e. between $\{\varphi_i^d\}_{i=1}^m$ and $\{\varphi_i^c\}_{i=1}^m$.

Finally from (4.1.2) and (4.1.3) we obtain the following formula for the preconditioner

$$P^{-1} = P^T Z_{h,b} P (R^T G_b P)^{-1} \tag{4.1.5}$$

where

$P \in \mathbb{R}^{m\times n} : \{\varphi_i\}_{i=1}^m \to \{\psi_i\}_{i=1}^n$ – realizes the mapping between div-conforming vector functions defined on the barycentric refinement and the div- and quasicurl-conforming vector functions on the dual mesh,

$Z_{h,b} \in \mathbb{R}^{m\times m}$ – Maxwell matrix on the barycentric refinement, i.e. evaluated on basis $\{\varphi_i\}_{i=1}^m$,

$R \in \mathbb{R}^{m\times n} : \{\varphi_i\}_{i=1}^m \to \{\Phi_i\}_{i=1}^n$ – maps from the div- conforming space on the barycentric refinement onto the div- conforming space on the original mesh,

$G_b \in \mathbb{R}^{m\times m}$ – is the mixed Gram matrix linking div- and curl- conforming vector functions on the barycentric refinement, i.e. $\{\varphi_i^d\}_{i=1}^m$ and $\{\varphi_i^c\}_{i=1}^m$.

## 4.2 Additional Matrices

### 4.2.1 Embedding matrix $P$

On the Fig.19 a typical mesh part is given. The coefficient corresponding to the reference edge is only influenced by the functions of the colored edges of the barycentric refinement. Arrows symbolize edge orientations used in the article [1].

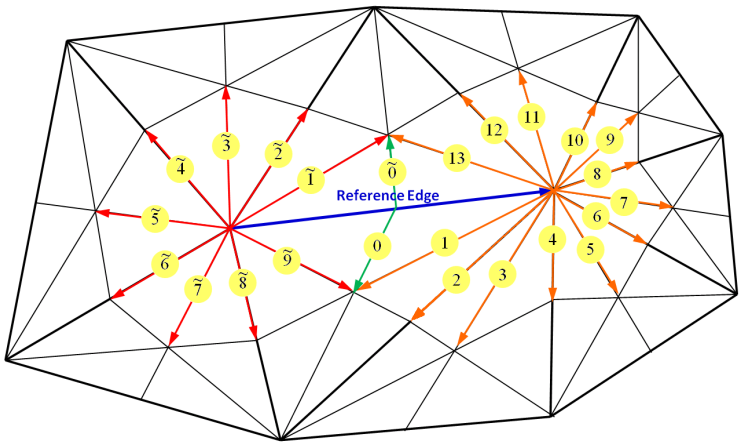



Figure 19: Edge orientations for $P$ matrix: thick lines denote original mesh, thin lines – barycentric refinement

The coefficients of this matrix are given in [1] and are as follows:

- for edges of the barycentric refinement that contain the first vertex of the reference edge the coefficients are assigned counterclockwise:

  $\tilde{c}_i^a = -\frac{N_1 - i}{2N_1}, \quad i = 1, ..., 2N_1 - 1$, where $N_1$ is the number of edges of the original mesh coming out of the first vertex of the reference edge;

- for edges of the barycentric refinement that contain the second vertex of the reference edge the coefficients are assigned counterclockwise:

  $c_i^a = \frac{N_2 - i}{2N_2}, \quad i = 1, ..., 2N_2 - 1$, where $N_2$ is the number of edges of the original mesh coming out of the second vertex of the reference edge;

- for edges of the barycentric refinement crossing the reference edge in the middle:

  $c_0^a = 1$ and $\tilde{c}_0^a = -1$.

In BETL edges are oriented differently. Thus we have to correct the orientation. In order to do this we need to take into account two additional orientations:

1. the mesh-induced edge orientation: from the node with lower index to the node with higher index;
2. the geometrical or BETL-induced edge orientation (Fig.20).

The mesh-induced orientation of red edges is always proper, since the additional nodes of the barycentric refinement always have higher indices by construction. Whereas green edges may have different orientation, that's why we need to correct their orientation by additional factor $\sigma_{m,i} = \pm 1$.

The geometrical or BETL-induced edge orientation is given by the order, in which vertices of the given triangle are stored. It is fixed for all elements of the barycentric refinement and is given on the Fig.20.

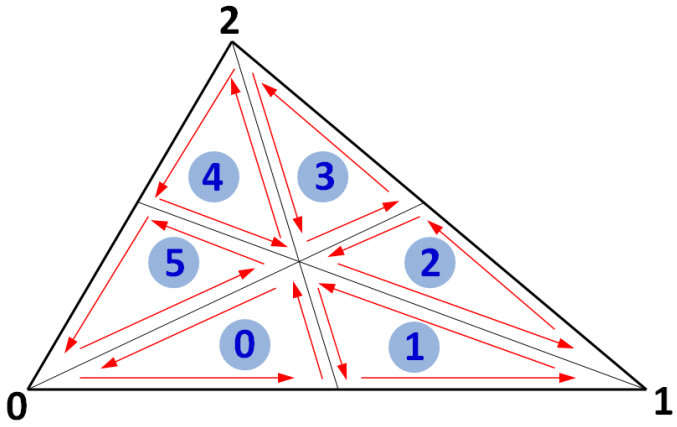



Figure 20: Geometry-induced edge orientation

Thus the final coefficients $c_i^{BETL}$ of the matrix $P$ will look this way:

$$c_i^{BETL} = \sigma_i^m \sigma_i^g c_i^a, \tag{4.2.1}$$

where $\sigma_i^m$ corrects the mesh-induced orientation and $\sigma_i^g$ corrects the geometry-induced orientation of the edge of the barycentric refinement.

### 4.2.2 Embedding matrix R

On the Fig.21 an element of the original mesh with six corresponding elements of the barycentric refinement is given. The reference edge is edge $AB$. The coefficient corresponding to this edge is only influenced by the functions of the red edges of the barycentric refinement. Arrows symbolize edge orientations used in the article [1].

$$\begin{array}{ll} c^a_{E_{FH}} = 1/6 & c^a_{E_{EH}} = -1/6 \\ c^a_{E_{AH}} = 1/3 & c^a_{E_{BH}} = -1/3 \\ c^a_{E_{AD}} = 1 & c^a_{E_{BD}} = 1 \end{array} \tag{4.2.2}$$

As in the case with $P$ matrix the coefficients of the $R$-matrix (4.2.2) are given in [1] and we only have to apply the orientation correction. Again some edges, i.e edges $AD$, $BD$, $AH$ and $BH$, have proper mesh-induced orientation, since the additional nodes of the barycentric refinement always

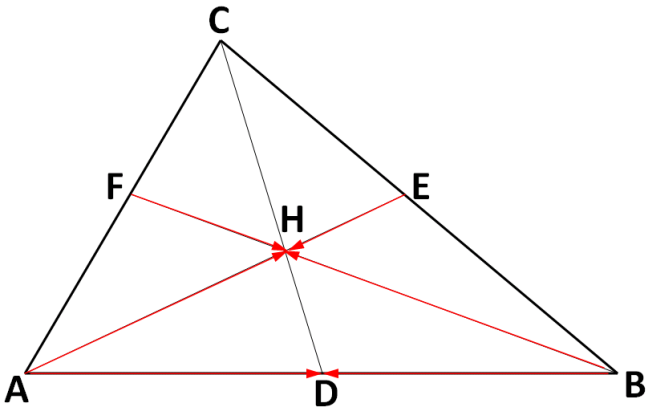



Figure 21: Edge orientation for $R$ matrix

have higher indices by construction. Whereas edges $FH$ and $HE$ can be arbitrary oriented.

And of course we have to account for geometry-induced orientation.

Finally we correct the original coefficients using formula:

$$c_{E_{ij}}^{BETL} = \sigma_{E_{ij}}^{m} \sigma_{E_{ij}}^{g} c_{E_{ij}}^{a}, \tag{4.2.3}$$

where $\sigma_{E_{ij}}^{m}$ corrects mesh-induced orientation and $\sigma_{E_{ij}}^{g}$ corrects geometry-induced orientation of the edge $E_{ij}$.

#### 4.2.3 Gram matrix

As a part of the preconditioner we need to invert the matrix $G$ (4.1.4).

BETL architecture does not require explicit matrix storage and all matrix operations are carried out using a matrix-vector product routine (*amux*) that is implemented for all matrix-types.

That is why we do not explicitly compute inverse matrices using additional existing packages, but use an iterative solver to implement an *amux* method for an inverse matrix. I.e. for each given vector $x$ our routine calculates $y = A^{-1}x$ using GMRes algorithm. Hence we create an implicit inverse matrix.

The termination criterion for the GMRes in this case is:

- either tolerance is less then the given one, i.e. $\epsilon < 10^{-5}$;
- or the maximum number of iterations is achieved $N_{max} = 1000$.

Using GMRes for matrix inversion proved to be very efficient in our case. The matrix that we are inverting – the mass matrix – is a regular sparse matrix. Moreover it is diagonally dominant, and thus Jacobi preconditioner improves the convergence of the method. As a result in all the models, that we have tested, the number of iterations needed for inversion was always $\approx 10$. A small number of iterations is sufficient, because the matrix $G$ is well conditioned independently of the resolution of the surface mesh.

Since this approach proved to be so efficient for our matrices, we decided to stay with it.

## 4.3 Results

### 4.3.1 Validation Model

As $Z_h$ matrix we are using the Maxwell matrix described above (3.2.8).

We are using a plane wave model, i.e.

$$\mathbf{u}(\mathbf{r}) = \mathbf{p} \exp(i\kappa \ \mathbf{d} \cdot \mathbf{r}) \tag{4.3.1}$$

where $\mathbf{d}$ – is the direction of wave propagation $\|\mathbf{d}\| = 1$, $\mathbf{p}$ – is polarization, i.e. the direction in which the electric field points $(\mathbf{p} \cdot \mathbf{d}) = 0$, $\kappa$ – is the wave number and $\mathbf{r}$ – is the position vector.

We validate our model on the Calderon identity:

$$\begin{pmatrix} -\left(\frac{1}{2}M_h + K_h\right) & V_h \\ \frac{1}{\kappa^2}V_h & -\left(\frac{1}{2}M_h - K_h\right)^* \end{pmatrix} \begin{pmatrix} u_D \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{4.3.2}$$

$V_h$ – is the Galerkin matrix of the Maxwell single layer potential defined in (3.2.8) evaluated on linear continuous div-conforming vector functions,

$K_h$ – is the Galerkin matrix of the Maxwell double layer potential evaluated on linear continuous div-conforming and linear continuous curl-conforming vector functions,

$M_h$ – is the mass matrix evaluated on linear continuous div-conforming and linear continuous curl-conforming vector functions.

We start with simple geometries (Fig.2). The tests parameters are: $\kappa = 1.0$, $\mathbf{p} = (-1, 1, 0)$ and $\mathbf{d} = (1, 1, 1)$ before normalization. We use default BETL ACA settings.

### 4.3.2 Test Models: Linear Vector Basis

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 192 | 81 | 8 |
| 768 | 140 | 8 |
| 3072 | 239 | 8 |
| 12288 | 403 | 8 |

Table 16: Sphere

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 381 | 127 | 11 |
| 1524 | 226 | 11 |
| 6096 | 385 | 11 |
| 24384 | 720 | 11 |

Table 17: Cylinder

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 576 | 124 | 12 |
| 2304 | 207 | 12 |
| 9216 | 360 | 13 |
| 36864 | 577 | 13 |

Table 18: Cube

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 216 | 129 | 10 |
| 864 | 228 | 11 |
| 3456 | 377 | 11 |
| 13824 | 609 | 12 |

Table 19: Fichera corner

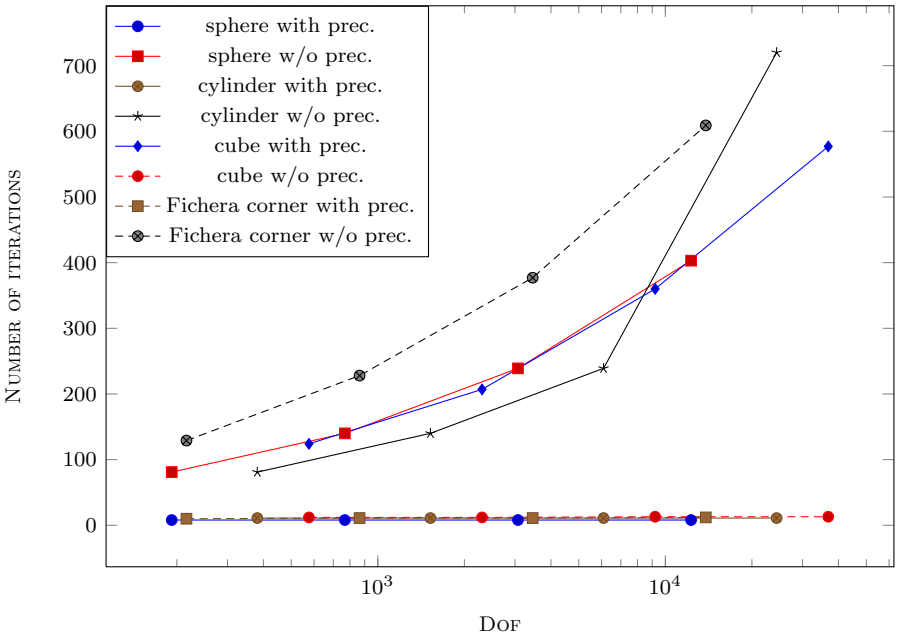



Figure 22: Test models, dual edge preconditioner

### 4.3.3 Complicated Geometry

In this case we use a two-discs model. It consists of two concentric discs with radius $R_d = 10$ and thickness $h = 2$, set apart on the distance $d = 6$ and connected by a cylinder with radius $R_c = 3$. All sharp corners are rounded with $r = 0.5$.

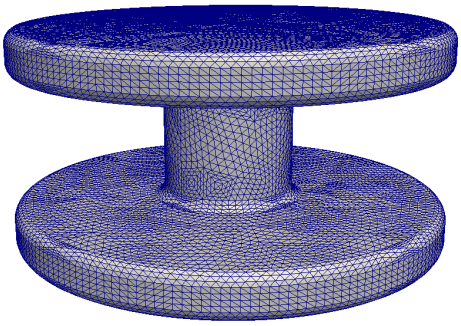


Figure 23: "Two discs" model: mesh

We apply a plane wave Dirichlet boundary conditions with $\mathbf{p} = (-1, 1, 0)$ and $\mathbf{d} = (\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$. We set $\kappa = 0.1$. And use the default ACA settings.

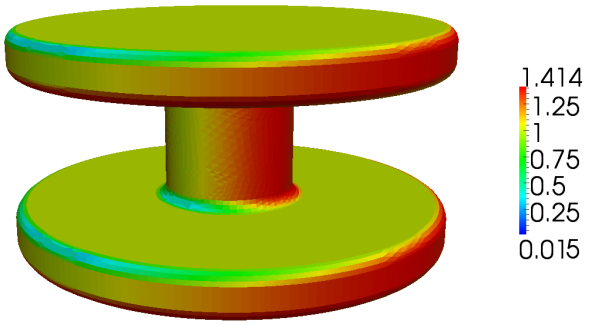



Figure 24: "Two discs" model: boundary conditions

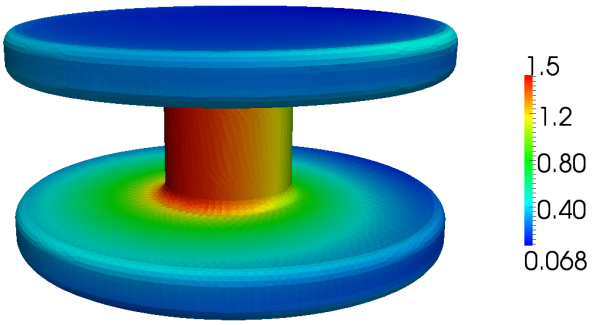



Figure 25: "Two discs" model: solution

| | GMRes w/o PC | GMRes with PC |
|---|---|---|
| # of iterations | 1120 | 24 |
| GMRes time, min | 174 | 6 |
| Assembling of preconditioner, min | - | 306 |
| Total time, min | 174 | 312 |

Table 20: "Two discs" model: dual edge preconditioner results

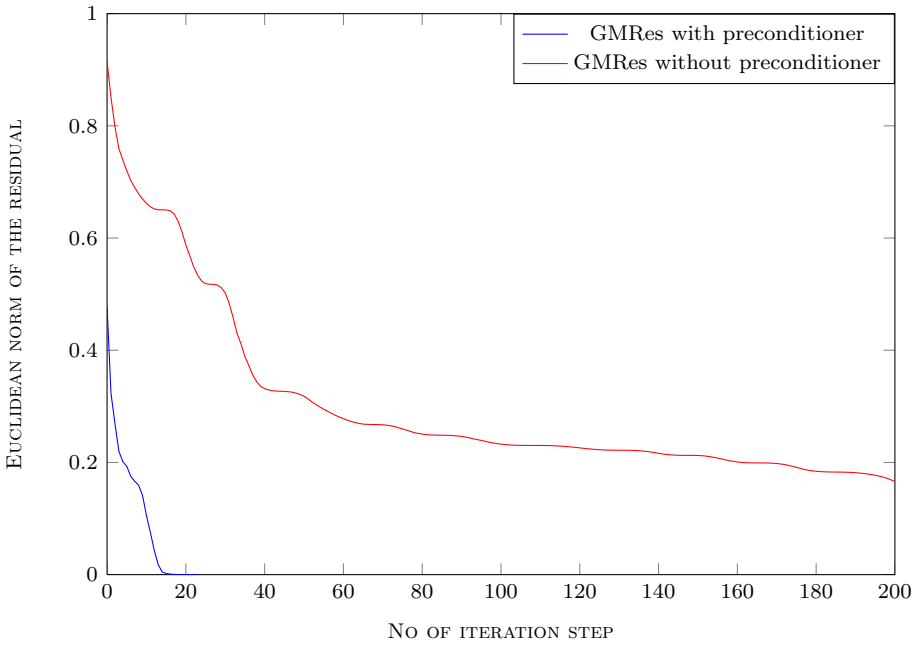



Figure 26: "Two discs" model: residuals

### 4.3.4 Low Frequency Test

To test the quality of the preconditioner we also ran a couple of tests for low frequency plane wave boundary conditions. As a test model we chose a sphere $R = 1$ with a moderate size mesh (12288 dofs). For the plane wave (4.3.1) we chose the following parameters $\mathbf{p} = (-1, 1, 0)$ and $\mathbf{d} = (1, 1, 1)$. And varied $\kappa \in [0.001, 1]$.

| $\kappa$ | 1 | 0.1 | 0.05 | 0.01 | 0.005 | 0.001 |
|---|---|---|---|---|---|---|
| # if iter. w/o prec | 403 | 437 | 420 | 307 | 294 | 225 |
| # if iter. with prec | 8 | 6 | 6 | 6 | 6 | 8 |

Table 21: Low frequency test, iteration numbers

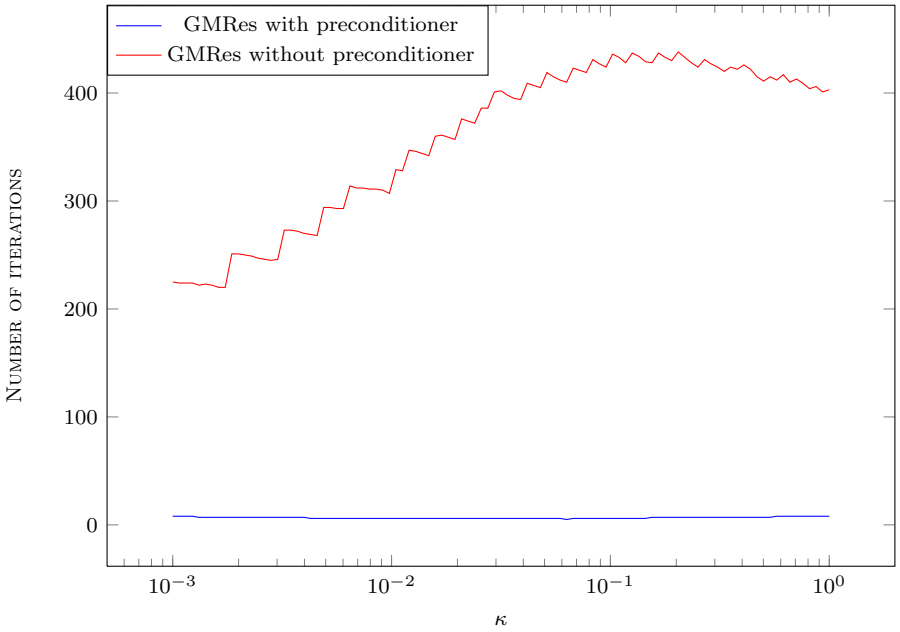
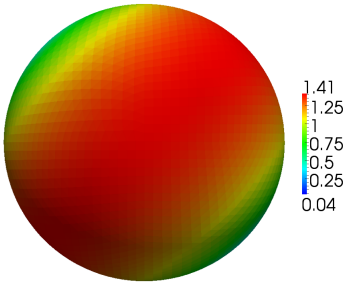
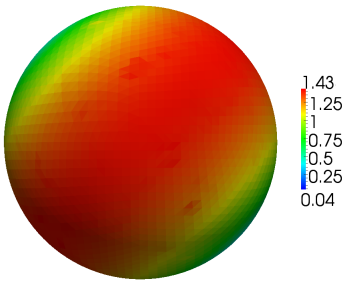



Figure 27: Low frequency test, iteration numbers

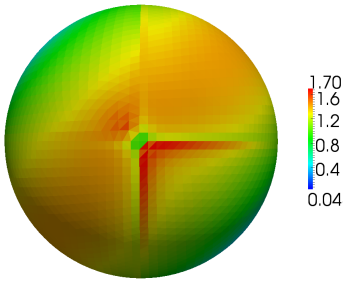
We observe the decrease in number of iterations for GMRes without preconditioner as $\kappa$ decreases. However the results without preconditioner are affected by huge errors.



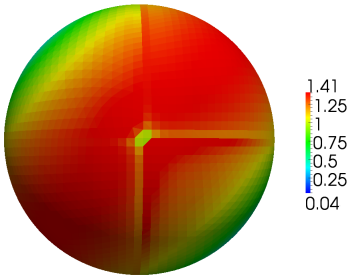

(a) theoretical solution




(b) solution with preconditioner




(c) solution w/o preconditioner




(d) solution w/o preconditioner, scaled to match theoretical solution

Figure 28: Low frequency test, $\kappa = 0.005$, $u_D$

# 5 Hierarchichal Edge Preconditioner

## 5.1 Preconditioner Formula

We follow the same strategy as in the scalar case (Sec.2.1). Only now we have eight basis functions on each triangle (3.1.20). And not just one, but the first three of them $\mathbf{e}_{ij}^{-}$ are the basis functions of the lowest order space available, i.e. of the linear space. Besides our bases are hierarchical by construction (Sec.3.1.1), that's why we don't need a special transformation $T$ in Sec.2.4.4.

Hence the formula for the hierarchical edge preconditioner simplifies to:

$$P^{-1} = {P_c}^T B_{cc} P_c + D_{ss}^{-1} \tag{5.1.1}$$

We have only implemented one case, when $V_h = R_1$ and $V_c = R_0$.

## 5.2 Additional Matrices

### 5.2.1 Block Diagonal Preconditioner

We want to create the matrix $D_{ss}^{-1}$ as in scalar case (2.4.3) – an inverse block diagonal structure. However in scalar case we dealt with discontinuous bases and discontinuous basis functions existed only on one triangle each and thus there we could work with blocks corresponding to different triangles separately. In case of vector functions and Maxwell equations we need to work with continuous bases. In this case neighboring triangles have common basis functions (Fig.29) and the corresponding blocks $D_{ss,loc}$ overlap and we cannot work with them independently.

The straightforward approach would be to assemble the complete block diagonal matrix $D_{ss}$ first, invert it using some external package and then zero out the lines corresponding to the lowest order functions by means of incidence matrices. However there is an easier and faster way.

We have chosen to apply *additive subspace correction*.

Let the discrete space $V_h \subset H^{-1/2}(\mathrm{div}_\Gamma, \Gamma)$ be a vector boundary element space with the basis functions described in Sec.3. It can be decomposed in the following way:

$$V_h = V_{low} + \sum_{k=1}^{N_e} V_{high,k}, \tag{5.2.1}$$

where $N_e$ is the number of elements in the mesh,

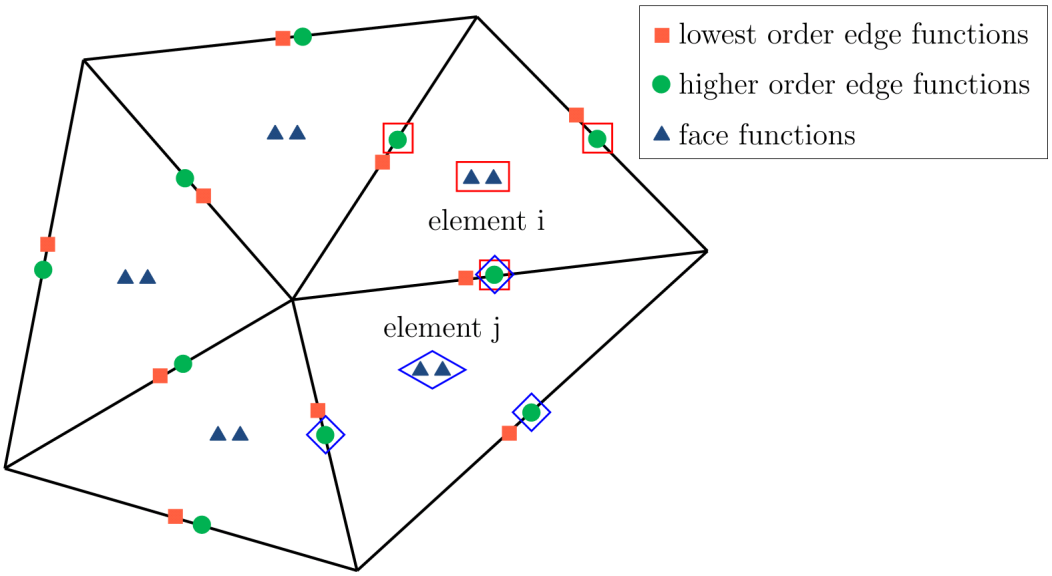



Figure 29: Dof's distribution in quadratic vector basis

$V_{low}$ consists of the functions $\mathbf{e}_{ij}^{-}$ from (3.1.20) corresponding to all edges of the mesh – marked orange on Fig.29,

$V_{high,k}$ for each $k$ consists of the remaining five functions $\mathbf{e}_{ij}^{+}$ and $\mathbf{f}_i$ on the $k$th triangle: on Fig.29 dofs corresponding to $V_{high,i}$ of the element $i$ are marked by a red frame and dofs corresponding to $V_{high,j}$ of the element $j$ are marked by a blue frame.

Clearly spaces $V_{high,k}$ corresponding to neighboring triangles overlap by a common higher order edge function, e.g. on Fig.29 elements $i$ and $j$ have a common higher order function that belongs both to $V_{high,i}$ and to $V_{high,j}$.

Now let our system matrix $A$ be a Galerkin matrix evaluated on the same basis. It has the block structure (2.1.6). And we want to build a block Jacobi preconditioner for the $A_{ss}$ part. Here is the algorithm we use.

**Additive Subspace Correction Method**

$\mathbf{c} = 0$
**for i=1:$\mathbf{N_e}$**

(1) **Evaluation of elements dof indices**
- calculate indices idx of the five vector functions on the $i$th triangle that comprise $V_{high,i}$

(2) **Extraction of block submatrix**
- extract the corresponding $5 \times 5$ submatrix $A_i$ from $A$: $A_i = A(\text{idx}, \text{idx})$

(3) **Matrix inversion**
- evaluate $\mathbf{w} = A_i^{-1}\mathbf{r}(\text{idx})$, where $\mathbf{r}$ is the residuals vector

(4) **Additive correction**
- add the result to the corresponding positions in the global vector $\mathbf{c}(\text{idx}) = \mathbf{c}(\text{idx}) + \mathbf{w}$

**end**

### 5.2.2 Projection Matrix $P_c$

The purpose of this matrix is to expand the lowest order preconditioner matrix $B_{cc}$, which has the size $n = \dim H_c$ to the size of the original matrix that we precondition, i.e to $N = \dim H_h$ by adding zero rows and columns. Thus $P_c$ is a $n \times N$ sparse matrix, that has the following structure:

$$(P_c)_{ij} = \begin{cases} 1 & \text{if } j\text{'s dof in } H_h \text{ corresponds to } i\text{'s dof in } H_c \\ 0 & \text{otherwise.} \end{cases} \tag{5.2.2}$$

## 5.3 Results

### 5.3.1 Validation Model

We are using the same equation, boundary conditions, plane wave parameters and models as in Sec.4.3.1. Only now in the Calderon identity

$$\begin{pmatrix} -\left(\frac{1}{2}M_h + K_h\right) & V_h \\ \frac{1}{\kappa^2}V_h & -\left(\frac{1}{2}M_h - K_h\right)^* \end{pmatrix} \begin{pmatrix} u_D \\ u_N \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \tag{5.3.1}$$

$V_h$ – is the Galerkin matrix of the Maxwell single layer potential defined in (3.2.8) evaluated on *quadratic* continuous div-conforming vector functions,

$K_h$ – is the Galerkin matrix of the Maxwell double layer potential evaluated on *quadratic* continuous div-conforming and *quadratic* continuous curl-conforming vector functions,

$M_h$ – is the mass matrix evaluated on *quadratic* continuous div-conforming and *quadratic* continuous curl-conforming vector functions.

### 5.3.2 Test Models

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 640 | 406 | 48 |
| 2560 | 779 | 51 |
| 10240 | 1021 | 51 |
| 40960 | 1330 | 55 |

Table 22: Sphere

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 1270 | 666 | 86 |
| 5080 | 1075 | 86 |
| 20320 | 1620 | 91 |
| 81280 | 2276 | 90 |

Table 23: Cylinder

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 1920 | 711 | 94 |
| 7680 | 1004 | 92 |
| 30720 | 1479 | 95 |
| 122880 | 2697 | 97 |

Table 24: Cube

| # of DOFs | w/o PC | with PC |
|---|---|---|
| 720 | 538 | 72 |
| 2880 | 1327 | 76 |
| 11520 | 1835 | 83 |
| 46080 | 2913 | 86 |

Table 25: Fichera corner

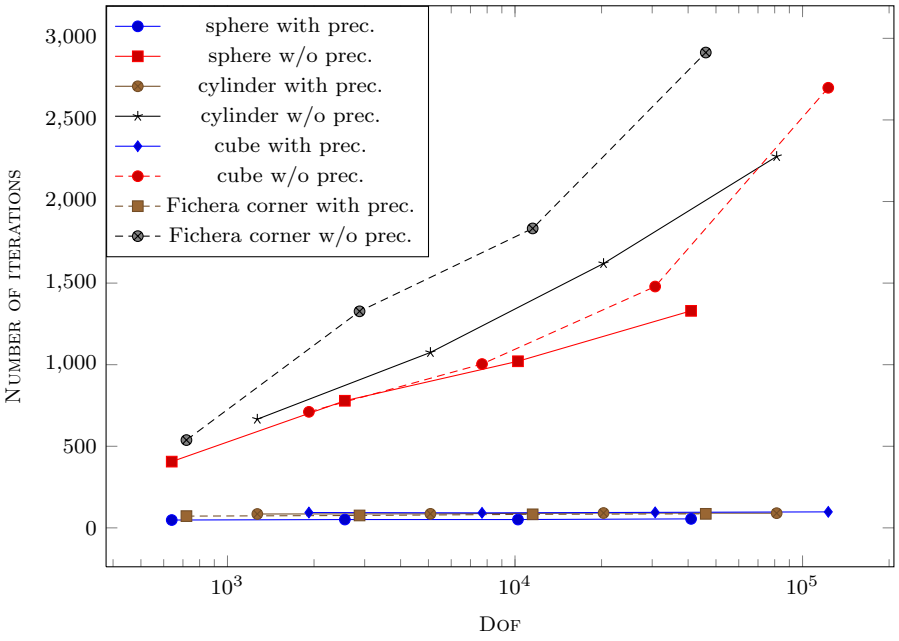



Figure 30: Test models: hierarchical edge preconditioner

#### 5.3.3 Complicated Geometry

We are again using the two discs model (Fig.23) with the same boundary conditions (Fig.24), plane wave parameters and ACA settings. $\kappa = 0.1$.

| | GMRes w/o PC | GMRes with PC |
|---|---|---|
| # of iterations | 3051 | 101 |
| GMRes time, min | 565 | 29 |
| Assembling of preconditioner, min | - | 367 |
| Total time, min | 565 | 396 |

Table 26: "Two discs" model: hierarchical edge preconditioner results

# 6 Conclusions

## 6.1 Quadratic Vector Basis

Based on the formulae from [7], 2nd order (quadratic) div- and curl- conforming vector bases on flat triangular elements were implemented. A special 2nd order accurate interpolation routine, different for edge and face functions, was also put in place.

## 6.2 Preconditioners

### 6.2.1 Scalar Hierarchical Preconditioner

We observe a significant decrease in iteration numbers, when we apply our preconditioner.These numbers remain almost constant, when we refine the mesh. However the time for the preconditioner construction is quite substantial. And on the tested models only in case of quadratic basis functions did the time for constructing preconditioner and solving the system with it was less than a simple GMRes. But for the tests for this paper we used sequential approach (i.e. just one processor). *OpenMP* environment will significantly speed the part of preconditioner construction. And in this case only CG/GMRes time will be important. Moreover if several launches of the model with different boundary conditions are planned, then using a preconditioner becomes highly effective.

### 6.2.2 Dual Vector Preconditioner

We generally observe the same behavior as in scalar hierarchical preconditioner case. The total time with preconditioner is still greater than that without it. But similarly *OpenMP* environment will definitely make time concerns less important.

However in this case the preconditioner has one more advantage. Not only does it decrease the number of iterations, but it also stabilizes the solution in low frequency case (Sec.4.3.4).

### 6.2.3 Vector Hierarchical Preconditioner

So far this model was the most laborious. We had a composite Maxwell BEM matrix, evaluated on the quadratic vector functions, and then we applied a dual Calderon preconditioner and wrapped it into a hierarchical struc-

ture. Thus the overall memory consumption and the total run-time for this model were enormous.

But in this case we finally do get a smaller total time for the case with preconditioner. Even without OpenMP.

## 6.3 Algorithmic Concerns

### 6.3.1 Inverse Block Diagonal Preconditioner

In order to create an inverse block-diagonal preconditioner for the surplus part of the hierarchical preconditioner we need to extract diagonal blocks of the original BEM matrix. But since BEM matrices in BETL are never explicitly stored, extraction would be very costly ($n^2$ complexity). That is why we recalculate these blocks once again inside the preconditioner routine. This approach results in linear complexity.

Of course in case of Laplace single layer potential for one diagonal block we only perform one block-integration, because this matrix is a simple BEM-matrix and basis functions used to evaluate it are discontinuous, thus the diagonal blocks are independent. Whereas for the Maxwell single layer potential we do several integrations of this type, because the basis is continuous and the matrix is a composition of two BEM-matrices and a sparse one. Still this approach is faster.

## 6.4 Matrix Inversion

As described in Sec.4.2.3 we use GMRes with Jacobi preconditioner to invert a sparse matrix. We used this approach for all the tests in this work and not only the average number of iterations was small ($\approx 10-15$), but also the average time for an inversion was significantly smaller than one matrix-vector multiplication for a BEM matrix. However for future models using one of the existing softwares for inversion of mass-matrices might be considered.

## 6.5 Additive Subspace Correction

In case of hierarchical vector preconditioner the diagonal blocks of the Maxwell matrix are interdependent, because the basis functions are continuous and exist on 2 triangles. Thus formally we would have to invert the block diagonal matrix $D_{ss}$ as a whole and zero out rows and columns corresponding to the lowest order functions using a sparse incidence matrix. Instead we opted for an additive approach (Sec.5.2.1).

### 6.6 Time for BEM Matrices Evaluation

In all the models, of course, most of the time is spent on creating the Galerkin matrices for the boundary integral operators. That is why we want, if possible, to minimize their quantity. Usually, when we talk about dual preconditioners, we have to calculate at least two BEM matrices: single layer potential matrix $V$ on the original mesh – the system matrix that we precondition, and a bigger hyper singular potential matrix $D$ on the barycentric refinement. However in case of simple Calderon preconditioner the system matrix can be obtained from the BEM matrix evaluated on the barycentric refinement via multiplication by an incidence matrix. This saves a lot of time. Unfortunately this approach is not possible in case of hierarchical preconditioners.

# A Description of C++ files

## A.1 Dual Mesh

### A.1.1 File: dual_mesh.hpp

Implements a new class *DualMesh*, which is inherited from the BETL *Mesh* class.

```
template < typename PARENT_MESH_T >
class DualMesh : public Mesh < BaryElement >
```

It is used to create a barycentric refinement of a given mesh.

Has no methods. The creation of the barycentric refinement is done in the class' constructor:

```
DualMesh ( const PARENT_MESH_T& parent_mesh )
```

Given a mesh in the BETL mesh format the constructor uses internal methods to create all the necessary geometrical entities and to make a barycentric refinement in the same mesh-format, i.e. in the ordinary BETL mesh-format.

## A.2 Dual Scalar Preconditioner

### A.2.1 File: preconditioner.hpp

Implements a new class *Preconditioner*. It assembles all the submatrices necessary for the creation of the preconditioner.

```
template < enum PARALLEL PAR, enum ACCELERATION ACC >
class Preconditioner
```

The class constructor has two arguments: original mesh and dofhandler of the piecewise constant basis functions on the original mesh.

```
DualPreconditioner( const parent_mesh_t& parent_mesh,
                    const dh_parent_const_t& dh_const).
```

Using these input variables the class creates dual mesh, evaluates all the necessary matrices and assembles them.

The class has an operator ():

```
template <class T>
void operator()(T* x) const.
```

For any given vector $\mathbf{x}$ it returns a vector $P^{-1}\mathbf{x}$, i.e. implements multiplication by the matrix $P^{-1}$.

### A.2.2 File: gauger.hpp

Implements a new class *HyperMatrixGauger*, which is inherited from the BETL *LaplaceGauger* class.

```
template < class MATRIX >
class HyperMatrixGauger: public LaplaceGauger< 1 >
```

It is used to perform the correction of the matrix of the hypersingular potential:

$$D_{hs} \to D_{hs} + \alpha \sum_{i=1}^{k} v_i \cdot v_i^T, \tag{A.2.1}$$

where $k$ – is a number of connected components in the mesh, $\vec{v}$ – is the gauger vector and $\alpha$ – is the scaling coefficient: $\alpha \approx 0.5(\lambda_{max} + \lambda_{min})$ , $\lambda_{max}, \lambda_{min}$ – are the minimal and the maximal eigenvalues of the matrix $D_{hs}$ [6].

The class constructor has two arguments: matrix $D_{hs}$ and the corresponding dofhandler.

```
template < class DOFHANDLER >
HyperMatrixGauger(const DOFHANDLER& dh, const MATRIX& A)
```

It supports the standard BETL matrix functions, i.e.

- ```
  template <class T>
  void amux ( T* x, T* y, T alpha, T beta, char op) const
  ```
- ```
  size_t GiveCols( ) const
  ```
- ```
  size_t GiveRows( ) const
  ```

## A.3 Hierarchical Scalar Preconditioner

### A.3.1 File: preconditioner.hpp

Implements a new class *Preconditioner*, that is used to assemble a preconditioner described in Sec.2.4.4.

```
template< typename LOW_ORDER_PRECOND_T,
          typename DOFHANDLER_LOW_ORDER_T,
          typename DOFHANDLER_HIGH_ORDER_T>
class Preconditioner
```

The class constructor has four arguments: preconditioner for the lowest order part, lowest order dofhandler, higher order dofhandler and an integrator.
```

```
template< typename INTEGRATOR_T >
Preconditioner( const LOW_ORDER_PRECOND_T& low_order_prec,
                const DOFHANDLER_LOW_ORDER_T&  dh_low,
                const DOFHANDLER_HIGH_ORDER_T& dh_high,
                const INTEGRATOR_T& integrator );
```

Dofhandlers are used to properly position preconditioners entries corresponding to dofs of lower and higher order. Integrator is used to recalculate diagonal blocks of the original matrix, which are used to create the inverse block diagonal part of the preconditioner.

Using these input variables the class creates all the necessary matrices and assembles them.

The class has an operator **()**:

```
template <class T>
void operator()(T* x) const.
```

For any given vector $\mathbf{x}$ it returns a vector $P^{-1}\mathbf{x}$, i.e. implements multiplication by the matrix $P^{-1}$.

### A.3.2 Files: T_local_functor.hpp and T_local_functor.cpp

Implements a new class *T_local*, that creates matrices $T_{loc}$ for the given higher order basis.

```
template< std::size_t DIM>
class T_local
```

So far just two matrices T_local< 3 > – (2.4.5), and T_local< 6 > – (2.4.8) were implemented.

The class constructor has no arguments.

```
T_local( )
```

The class has two standard methods: **giveMatrix( )** and operator **(i,j)**:

```
const_reference giveMatrix( ) const

double operator()( const std::size_t row,
                   const std::size_t col ) const .
```

### A.3.3 File: hierarchical_basis.hpp

Implements a new class *HierarchicalBasis*, which is inherited from the BETL *RootSparseOperator* class.

```
template< typename DOFHANDLER_HIGH_ORDER_T >
class HierarchicalBasis : public betl::RootSparseOperator
```

It is used to assemble a global $T$ matrix, that will split the basis into lowest order and surplus parts (2.2.2).

The class constructor has only one argument: higher order dofhandler.

```
HierarchicalBasis( const DOFHANDLER_HIGH_ORDER_T& dh_high );
```

The class has only one method **compute**:

```
template< typename T_LOCAL >
void compute( T_LOCAL t_local );
```

This functions creates small blocks of $T_{loc}$ (2.2.1) from t_local and spreads them on the global positions.

#### A.3.4 File: low_frequency_extractor.hpp

Implements a new class *LowFrequencyExtractor*, which is inherited from the BETL *RootSparseOperator* class.

```
template< typename DOFHANDLER_LOW_ORDER_T,
          typename DOFHANDLER_HIGH_ORDER_T >
class LowFrequencyExtractor : public
betl::RootSparseOperator
```

It is used to assemble an incidence matrix $P_c$ (2.4.1).

The class constructor has two arguments: lower and higher order dofhandlers.

```
LowFrequencyExtractor( const DOFHANDLER_LOW_ORDER_T& dh_low,
                       const DOFHANDLER_HIGH_ORDER_T& dh_high );
```

The class has only one method **compute**:

```
void compute( )
```

that assembles the matrix according to the formula (2.4.1).

#### A.3.5 File: inverse_block_diagonal.hpp

Implements a new class *InverseBlockDiagonalOperator*, which is inherited from the BETL *RootSparseOperator* class.

```
template< typename DOFHANDLER_HIGH_ORDER_T >
class InverseBlockDiagonalOperator : public
betl::RootSparseOperator
```

It is used to assemble the block diagonal part of the preconditioner $D_{ss}^{-1}$ (2.4.9).

The class constructor has only one argument: higher order dofhandler.

```
InverseBlockDiagonalOperator
    ( const DOFHANDLER_HIGH_ORDER_T& dh );
```

The class has one method **compute**:

```
template< typename INTEGRATOR_T >
void compute( const INTEGRATOR_T& integrator );
```

This functions creates the matrix $D_{ss}^{-1}$ as described in Sec.2.4.3.

## A.4 Dual Vector Preconditioner

### A.4.1 File: preconditioner.hpp

Implements a new class *DualEdgePreconditioner*.

```
template < enum ACCELERATION ACC, enum PARALLEL PAR >
class DualEdgePreconditioner
```

The class constructor has three arguments: original mesh, dofhandler of the linear continuous vector basis functions on the original mesh and a parser with ACA settings.

```
DualEdgePreconditioner
               ( const parent_mesh_t&       mesh,
                 const parent_dofhandler_t& parent_dh,
                 const settings_parser_t& parser )
```

Using these input variables the class creates dual mesh, evaluates all the necessary matrices and assembles them.

The class has a function **compute**:

```
void compute( const complex_t kappa )
```

that assembles all necessary submatrices.

This preconditioner class is a bit different from the others. Since in this case as a preconditioner we use a matrix of the same type as a system matrix, we can save time by calculating it only once. Thus the *DualEdgePreconditioner* class has the following methods:

```
preconditioner_reference givePreconditioner( )

    matrix_reference       giveSystemMatrix( )
```

The system matrix and the preconditioner obtained this way have all the standard functions, i.e. **amux** and **operator()** correspondingly.

### A.4.2 File: coupling_matrix.hpp

Implements a new class *coupling_operator*, that assembles $R$ incidence matrix (Sec.4.2.2). It is inherited from *RootSparseOperator* class.

```
template< typename DUAL_DOFHANDLER_T,
          typename PARENT_DOFHANDLER_T >
class coupling_operator: public
betl::RootSparseOperator
```

The class constructor has two arguments: dofhandlers of the linear continuous vector basis functions on the original mesh and on the barycentric refinement.

```
coupling_operator( const dual_dh_t&   dual_dh,
                   const parent_dh_t& parent_dh )
```

The class has a function **compute**:

```
void compute( const complex_t kappa )
```

that assembles the $R$ matrix.

### A.4.3 File: averaging_matrix.hpp

Implements a new class *averaging_operator*, that assembles $P$ incidence matrix (Sec.4.2.1). It is inherited from *RootSparseOperator* class.

```
template< typename DUAL_DOFHANDLER_T,
          typename PARENT_DOFHANDLER_T >
class averaging_operator: public
betl::RootSparseOperator
```

The class constructor has two arguments: dofhandler of the linear continuous vector basis functions on the original mesh and on the barycentric refinement.

```
averaging_operator( const dual_dh_t&   dual_dh,
                    const parent_dh_t& parent_dh )
```

The class has a function **compute**:

```
void compute( const complex_t kappa )
```

that assembles the $P$ matrix.

## A.5 Hierarchical Vector Preconditioner

### A.5.1 File: preconditioner.hpp

Implements a new class *Preconditioner*.

It is used to assemble a preconditioner for a given mesh.

The class constructor has six arguments: the first three are as in the scalar hierarchical preconditioner case – preconditioner for the lowest order part, lowest order dofhandler, higher order dofhandler, and then we have two integrators and a scalar, because this time the system matrix is a composite one, i.e. created on base of two integrators and with a parameter $\kappa$.

```
template< typename INTEGRATOR_T1, typename INTEGRATOR_T2 >
Preconditioner( const LOW_ORDER_PRECOND_T& low_precond,
                const DOFHANDLER_LOW_ORDER_T&  dh_low,
                const DOFHANDLER_HIGH_ORDER_T& dh_high,
                const INTEGRATOR_T1& integrator1,
                const INTEGRATOR_T2& integrator2,
                const complex_t scalar )
```

Dofhandlers are used to properly position preconditioners entries corresponding to dofs of lower and higher order. Integrators and scalar are used to recalculate diagonal blocks of the original matrix, which are used to create the block diagonal part of the preconditioner.

Using these input variables the class creates all the necessary matrices and assembles them.

The class has an operator ():

```
template <class T>
void operator()(T* x) const.
```

For any given vector $\mathbf{x}$ it returns a vector $P^{-1}\mathbf{x}$, i.e. implements multiplication by the matrix $P^{-1}$.

### A.5.2 File: low_frequency_extractor.hpp

Implements a new class *LowFrequencyExtractor*, which is inherited from the BETL *RootSparseOperator* class.

```
template< typename DOFHANDLER_LOW_ORDER_T,
          typename DOFHANDLER_HIGH_ORDER_T,
          typename T >
class LowFrequencyExtractor : public
betl::RootSparseOperator
```

It is used to assemble an incidence matrix $P_c$ (5.2.2).

The class constructor has two arguments: lower and higher order dofhandlers.

```
LowFrequencyExtractor
                    ( const DOFHANDLER_LOW_ORDER_T& dh_low,
                      const DOFHANDLER_HIGH_ORDER_T& dh_high );
```

The class has only one function **compute**:

```
void compute( )
```

that assembles the matrix according to the formula (5.2.2).

### A.5.3 File: inverse_block_diagonal.hpp

Implements a new class *InverseBlockDiagonalOperator*, which is inherited from the BETL *RootSparseOperator* class .

```
template < typename DOFHANDLER_HIGH_ORDER_T,
           typename T >
class InverseBlockDiagonalOperator
```

It is used to assemble the block diagonal part of the preconditioner $D_{ss}^{-1}$ (Sec.5.2.1).

The class constructor has only one argument: higher order dofhandler.

```
InverseBlockDiagonalOperator
             ( const DOFHANDLER_HIGH_ORDER_T& dh );
```

The class has only one method **compute**:

```
template< typename INTEGRATOR_T, typename EDGE_BEM_T >
void compute( const INTEGRATOR_T& integrator,
              const EDGE_BEM_T& edge_bem, T scalar )
```

This functions creates the matrix $D_{ss}^{-1}$ as described in Sec.2.4.3.

## A.6 Other Files

### A.6.1 File: inverse_matrix.hpp

Implements a new class *InverseMatrixWithPreconditioner* with proper inheritance from *MatrixExpression* class.

```
template <class MATRIX_T, class PRECONDITIONER_T>
class InverseMatrixWithPreconditioner:
public betl::linalg::MatrixExpression
                    < InverseMatrixWithPreconditioner
                    < MATRIX_T, PRECONDITIONER_T > >
```

This class inverts a given matrix, using a given preconditioner.

It is used to invert a matrix by means of the GMRes algorithm (Sec.4.2.3).

The class constructor has the following structure:

```
InverseMatrixWithPreconditioner(const MATRIX_T& A,
                                const PRECONDITIONER_T* P,
                                int GMRes_max_iter,
                                double GMRes_tolerance)
```

Since the result of the inversion is again a matrix, it supports the standard BETL matrix functions, i.e.

- `template <class T>`
  `void amux ( T* x, T* y, T alpha, T beta, char op) const`
- `size_t GiveCols( ) const`
- `size_t GiveRows( ) const`

The file also includes class specializations for the existing Jacobi preconditioner and for the case, when no preconditioner is used.

### A.6.2 File: preconditioner_from_matrix.hpp

Implements a new class *PreconditionerFromMatrix*.

```
template< typename MATRIX_T >
class PreconditionerFromMatrix
```

It is a simple wrapper that for a given matrix $A$ implements the standard preconditioner function **compute**:

```
template< typename T >
void operator()( T* x ) const
```

### A.6.3 File: chain_preconditioner.hpp

Implements a new class *ChainPreconditioner*.

```
template< typename PRECONDITIONER_A_T,
          typename PRECONDITIONER_B_T >
class ChainPreconditioner {
```

This class consequently applies two given preconditioners.

It is a simple wrapper that implements the standard preconditioner function **compute**:

```
template< typename T >
void operator()( T* x ) const
```

## References


[1] F.P.Andriulli et al., *A Multiplicative Calderon Preconditioner for the Electric Field Integral Equation*. IEEE Transactions on Antennas and Propagation Vol.56, No.8, August 2008, 2398-2409.

[2] A. Buffa, M. Costabel, and C. Schwab, *Boundary element methods for Maxwell's equations on non-smooth domains*. Numer. Math. 92 (2002), 679-710.

[3] R.Hiptmair, *Operator Preconditioning*. Computers and mathematics with Applications 52 (2006) 699-706.

[4] L.Kielhorn, *BETL Documentation*. SAM - Seminar for Applied Mathematics, ETH Zurich.

[5] P.E.Meury, *Stable Finite Element Boundary Element Galerkin Schemes for Acoustic and Electromagnetic Scattering*. Diss. No. 17320, ETH Zurich.

[6] G.Of, *BETI–Gebietszerlegungsmethoden mit schnellen Randelementverfahren und Anwendungen*. Dissertation, Institut für Angewandte Analysis und Numerische Simulation, Universität Stuttgart, 2006.

[7] J.Schöberl, S.Zaglmayr, *High order Nédélec elements with local complete sequence properties*. COMPEL: The International Journal for Computation and Mathematics in Electrical and Electronic Engineering, Vol. 24, Iss: 2 pp. 374 - 384

[8] O.Steinbach, *Artificial Multilevel Boundary Element Preconditioners*. PAMM, Volume 3, Issue 1, pages 539–542, December 2003.