ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Algebraic Multigrid for Regularized Magnetostatics

Bachelor Thesis

Cedric Münger

July 2017

Supervisor: Prof. Dr. Ralf Hiptmair
Department of Mathematics, ETH Zürich

Abstract

In this Bachelor thesis the 2D magnetostatic problem

$$\mathbf{curl}_{2D}\, \mu(\boldsymbol{x})\, \mathrm{curl}_{2D}\, \boldsymbol{a} = \boldsymbol{j}, \quad \mathrm{div}\, \boldsymbol{a} = 0 \text{ in } \Omega, \quad \boldsymbol{a} \times \boldsymbol{n} = 0 \text{ on } \partial\Omega$$

is discretized using lowest order edge elements.

The arising linear system is solved with the 'classical' algebraic multigrid method introduced by J. W. Ruge and K. Stüben. The convergence of the AMG method is studied on regularly and locally refined triangular meshes and compared to the convergence if AMG is applied to a Laplace problem.

For the assembly of the finite element matrix the BETL library is used. As Solver BoomerAMG, the implementation of algebraic multigrid in the HYPRE library, is used. Validations of the implementation as well as the conducted numerical experiments are reported and discussed.

Contents

Chapter 1

Introduction

1.1 Problem formulation

On a bounded domain $\Omega \subset \mathbb{R}^2$ with trivial topology we consider the boundary value problem

$$\mathbf{curl}_{2D}\, \mu(\boldsymbol{x})\, \mathrm{curl}_{2D}\, \boldsymbol{a} = \boldsymbol{j}, \quad \mathrm{div}\ \boldsymbol{a} = 0 \text{ in } \Omega, \quad \boldsymbol{a} \times n = 0 \text{ on } \partial\Omega \tag{1.1}$$

with $\mu \in L^\infty(\Omega)$ uniformly positive. We assume $\mathrm{div}\ \boldsymbol{j} = 0$. This is the 2-dimensional vector potential formulation for a static magnetic field.

1.1.1 Variational formulation

Following the approch in [1], we can recast (1.1) as

$$\begin{cases} \text{find } \boldsymbol{a} \in \boldsymbol{H_0}(\boldsymbol{curl}, \Omega) \text{ that minimzes } J(\boldsymbol{a}) = \int_\Omega \mu(x)\, \mathrm{curl}_{2D}\, \boldsymbol{a} \cdot\ \mathrm{curl}_{2D}\, \boldsymbol{a} \mathrm{d}\boldsymbol{x} - \int_\Omega \boldsymbol{j} \cdot \boldsymbol{a} \mathrm{d}\boldsymbol{x} \\ \text{subjected to } \int_\Omega \boldsymbol{a} \cdot\ \mathbf{grad}\ \psi \mathrm{d}\boldsymbol{x} = 0 \quad \forall \psi \in H_0^1(\Omega) \end{cases} \tag{1.2}$$

where we used integration by parts and Green's formula.

$$\int_\Omega \mathrm{curl}_{2D}\, \boldsymbol{u} \cdot \varphi \mathrm{d}\boldsymbol{x} = \int_\Omega \boldsymbol{u} \cdot\ \mathbf{curl}_{2D}\, \varphi \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (\boldsymbol{u} \times \boldsymbol{n}) \cdot \varphi \mathrm{d}s$$

By introducing the Lagrange multiplier $\varphi \in H_0^1(\Omega)$ we can transform the problem into finding $(\boldsymbol{a}, \varphi) \in \boldsymbol{H}_0(\mathbf{curl}, \Omega) \times H_0^1(\Omega)$ for the Lagrange functional

$$L(\boldsymbol{a}, \varphi) = \int_\Omega \mu(x)\, \mathrm{curl}_{2D}\, \boldsymbol{a} \cdot\ \mathrm{curl}_{2D}\, \boldsymbol{a} \mathrm{d}\boldsymbol{x} - \int_\Omega \boldsymbol{j} \cdot \boldsymbol{a} \mathrm{d}\boldsymbol{x} + \int_\Omega \boldsymbol{a} \cdot\ \mathbf{grad}\ \varphi \mathrm{d}\boldsymbol{x}$$

We use standard calculus of variation techniques to arrive at the problem.
Find $\boldsymbol{a} \in \boldsymbol{H}(\mathbf{curl}, \Omega)$ and $\varphi \in H_0^1(\Omega)$ such that

$$\begin{aligned} \int_\Omega \mu(x)\, \mathrm{curl}_{2D}\, \boldsymbol{a} \cdot\ \mathrm{curl}_{2D}\, \boldsymbol{v} \mathrm{d}\boldsymbol{x} + \int_\Omega \boldsymbol{v} \cdot\ \mathbf{grad}\ \varphi \mathrm{d}\boldsymbol{x} &= \int_\Omega \boldsymbol{j} \cdot \boldsymbol{v} \mathrm{d}\boldsymbol{x} \qquad \forall \boldsymbol{v} \in \mathbf{H}_0(\mathbf{curl}, \Omega) \\ \int_\Omega \boldsymbol{a} \cdot\ \mathbf{grad}\ \psi \mathrm{d}\boldsymbol{x} &= 0 \qquad \forall \psi \in H_0^1(\Omega) \end{aligned}$$

We can employ Lemma 3 from [4], then we have for $\varphi = 0$, if $\operatorname{div} \boldsymbol{j} = 0$. We obtain:
Find $\boldsymbol{a} \in \boldsymbol{H}(\mathbf{curl}, \Omega)$ and $\varphi \in H_0^1(\Omega)$ such that

$$\begin{aligned} \int_\Omega \mu(x) \operatorname{curl}_{2D} \boldsymbol{a} \cdot \operatorname{curl}_{2D} v \mathrm{d}\boldsymbol{x} + \int_\Omega v \cdot \mathbf{grad}\, \varphi \mathrm{d}\boldsymbol{x} &= \int_\Omega \boldsymbol{j} \cdot v \mathrm{d}\boldsymbol{x} \qquad \forall v \in \mathbf{H}_0(\mathbf{curl}, \Omega) \\ \int_\Omega \boldsymbol{a} \cdot \mathbf{grad}\, \psi \mathrm{d}\boldsymbol{x} - \int_\Omega \varphi\psi \mathrm{d}\boldsymbol{x} &= 0 \qquad \forall \psi \in H_0^1(\Omega) \end{aligned}$$

1.1.2 Discretization

We employ finite element Galerkin discretization based on lowest order conforming finite element spaces. For $\mathbf{H}_0(\mathbf{curl}, \Omega)$ we use lowest order edge elements (Whitney-1 forms) [see Chapter 2] and for $H_0^1(\Omega)$ we use linear Lagrangian elements with standard basis functions. This leads to a linear system of following block form.

$$\begin{pmatrix} A & B \\ B^T & -D \end{pmatrix} \begin{pmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\varphi} \end{pmatrix} = \begin{pmatrix} \boldsymbol{i} \\ 0 \end{pmatrix}$$

By block Gauss elimination we arrive at a Schur complement system

$$\tilde{A}\boldsymbol{\alpha} = (A + BD^{-1}B^T)\boldsymbol{\alpha} = \boldsymbol{i}. \tag{1.3}$$

We use vertex based quadrature to approximate the L^2 inner product $\int_\Omega \varphi\psi \mathrm{d}\boldsymbol{x}$, such that the matrix D is diagonal. Then the inversion of D is trivial and therefore the system matrix $\tilde{A}$ remains sparse.

Chapter 2

Edge elements

The basis functions of Whitney 1-forms on a triangle in 2D are $\boldsymbol{b}_{i,j} = \lambda_i \, \mathbf{grad} \, \lambda_j - \lambda_j \, \mathbf{grad} \, \lambda_i$ [2], where i, j are indices of the vertices and λ_i is the barycentric coordinate function associated with vertex i. In a finite element formulation these basis functions serve as local shape functions. Since these basis functions are associated with the edge of the triangle, it becomes obvious why they are also called edge elements.

2.1 Edge element shape function

On a triangle with vertices $\boldsymbol{a}_1, \boldsymbol{a}_2, \boldsymbol{a}_3$ we denote the edge from $\boldsymbol{a}_i$ to $\boldsymbol{a}_{i+1}$ as $\boldsymbol{e}_i$. We associated the local shape function $\boldsymbol{b}_{i,i+1} = \lambda_i \, \mathbf{grad} \, \lambda_{i+1} - \lambda_{i+1} \, \mathbf{grad} \, \lambda_i$ with the the edge e_i.

On the reference triangle with vertices $\boldsymbol{a}_1 = [0,0]^T, \boldsymbol{a}_2 = [0,1]^T$ and $\boldsymbol{a}_3 = [1,1]^T$ the basis functions are:

$$\boldsymbol{b}_{1,2} = \begin{pmatrix} 1-y \\ x-1 \end{pmatrix}$$

$$\boldsymbol{b}_{1,2} = \begin{pmatrix} -y \\ x \end{pmatrix}$$

$$\boldsymbol{b}_{1,2} = \begin{pmatrix} -y \\ x-1 \end{pmatrix}$$

The curl of each basis function is constant like the gradient for linear Lagrange elements. On the reference triangle the curls of local shape function are

$$\mathrm{curl}_{2\mathrm{D}} \, \boldsymbol{b}_{1,2} = 2$$

$$\mathrm{curl}_{2\mathrm{D}} \, \boldsymbol{b}_{2,3} = 2$$

$$\mathrm{curl}_{2\mathrm{D}} \, \boldsymbol{b}_{3,1} = 2.$$

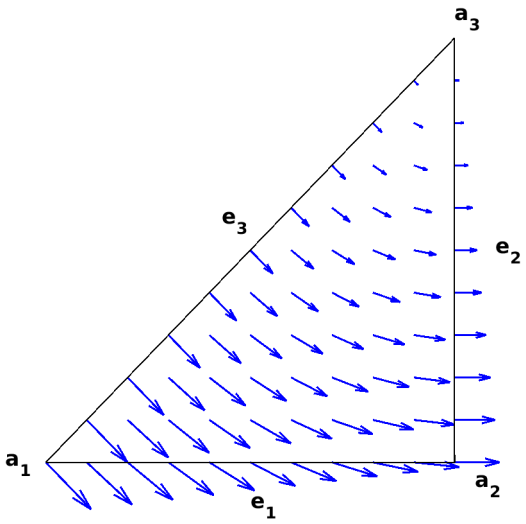

Figure 2.1: Local shape function $\boldsymbol{b}_{1,2}$ for reference triangle

Orientation

Another issue we have to address is the orientation of the edges. The local orientation of the edges is given by $\boldsymbol{e}_1 : \boldsymbol{a}_1 \to \boldsymbol{a}_2, \boldsymbol{e}_2 : \boldsymbol{a}_2 \to \boldsymbol{a}_3, \boldsymbol{e}_3 : \boldsymbol{a}_3 \to \boldsymbol{a}_1$. But in a triangulation it can happen that an edge has two opposite local orientation with respect to the two adjacent triangles. Therefore a global orientation of the edges has to be enforced and is used in the assembly of the local element matrices.

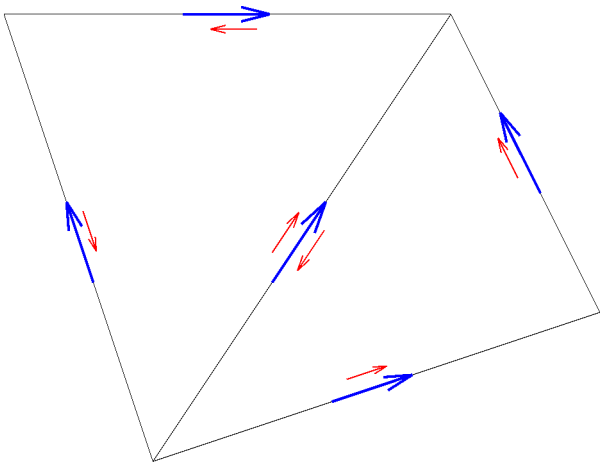

Figure 2.2: The local orientation of edges (red) may not be the same as the global orientation (blue) of the mesh.

2.1.1 Transformation

When using edge elements, one cannot simply use the affine mapping $F_K(\hat{x}) = B_K\hat{x} + a_K$ from the reference element $\hat{K}$ to the element K. Here a slightly different mapping [6] is

used:

$$\boldsymbol{b}(x) = (DF_K^{-T}\hat{\boldsymbol{b}}) \circ F_K^{-1}(x)$$

where DF_K is the Jacobian of the element map F_K. In the case of a affine mapping the is Jacobian DF_K is equal to B_K. This mapping is also known as the covariant Piola transformation.

Transformation of the curl

The curl transforms according to Theorem 4 in [6] to

$$\operatorname{curl}_{2D} \boldsymbol{b} = \det B_K^{-1} \operatorname{curl}_{2D} \hat{\boldsymbol{b}}. \tag{2.1}$$

From this we get further

$$\int_K \operatorname{curl}_{2D} \boldsymbol{b}_i \operatorname{curl}_{2D} \boldsymbol{b}_j dx = \int_{\hat{K}} \operatorname{curl}_{2D} \hat{\boldsymbol{b}} \cdot \operatorname{curl}_{2D} \hat{\boldsymbol{b}} |\det DF|^{-1} dx.$$

Chapter 3

Algebraic multigrid method

The algebraic multigrid method AMG is an iterative multigrid method. The main idea of a multigrid method is to relax the finest grid recursively to a coarsest grid, where the cost of solving is low. And then interpolate a correction back to the finest grid to achieve the desired accuracy. In difference to other multigrid methods the algebraic needs no information about the grid to build the hierarchy of the different levels. This information is determined automatically from the system matrix.

3.1 Classical AMG

There are different variations of the algebraic multigrid method. The main idea stays the same but the coarsening and smoothing techniques may vary. Here we focus on the technique presented by Ruge and Stüben [5].

Before the actual solving process the hierarchy of the grid has to be constructed. In this setup phase the points of the coarser grid are chosen in a specific way described in 3.1.1 and the interpolation operator I_{m+1}^{m} between the finer and the coarser grid is built.

Listing 3.1: Setup phase of algebraic multigrid

Step 1:	Set $m = 1$ and $A^1 = A$.
Step 2:	Choose coarse grid Ω^{m+1} and define I_{m+1}^{m}
Step 3:	Set $I_{m}^{m+1} = (I_{m+1}^{m})^T$ and $A^{m+1} = I_{m}^{m+1} A^m I_{m+1}^{m}$
Step 4	If Ω^{m+1} small enough stop. Otherwise set m=m+1 and go to Step 2

3.1.1 Coarsening

In the coarsening process the points of the gird are split into two sets C and F, where C is the set of the points of the coarser grid. Before we can start dividing the points, we introduce the notion of strongly connected. A point i is strongly connected to j, if $-a_{ij} \geq \theta \max_{l \neq i}\{-a_{il}\}$ with $0 < \theta \leq 1$ (usually 0.25). To get good results the splitting into C and F-Points has to fulfil two properties.

1. For each $i \in F$, each strongly connected point to i should be either in C or should be strongly connected to at least one point in C.
2. C should be a maximal subset of all points such that no two C-Points are strongly connected.

To ensure these properties the coarsening process has two steps. In the first step the *C*-points are distributed over the grid in compliance with the second property. And in the second step F-points are tested with the first property and if needed additional *C*-points are added.

Chapter 4

Implementation

4.1 BETL assembly

4.1.1 Assembly of curl curl matrix

For evaluation of the integral $\int_K \mu \ \mathbf{curl}_{2\mathrm{D}} \boldsymbol{a} \cdot \ \mathbf{curl}_{2\mathrm{D}} v dx$ we distinguish two cases. The first is that μ is constant, then we can derive analytical expression as shown below. But if we are unlucky and μ isn't constant we have to use numerical quadrature to evaluate the integral. For this we can use the transformation formulas introduced in 2.1.1.

Constant μ

For now we assume μ is constant. Then we have

$$\int_K \mu \ \mathrm{curl}_{2\mathrm{D}} \boldsymbol{b}_{ij} \cdot \ \mathrm{curl}_{2\mathrm{D}} \boldsymbol{b}_{kl} \mathrm{d}\boldsymbol{x} = 4\mu(\ \mathbf{grad}\ \lambda_i \times \ \mathbf{grad}\ \lambda_j) \cdot (\ \mathbf{grad}\ \lambda_l \times \ \mathbf{grad}\ \lambda_k)|K| = \frac{\mu}{|K|}$$

here we use that $\mathrm{curl}_{2\mathrm{D}} \boldsymbol{b}_{ij} = 2\ \mathbf{grad}\ \lambda_i \times \ \mathbf{grad}\ \lambda_j = const.$ and $\mathbf{grad}\ \lambda_i \times \ \mathbf{grad}\ \lambda_j = \frac{1}{2|K|}$

Non constant μ

For $\mu(x) \neq$ const. we have to use numerical quadrature. Here we can use the idea of parametric finite elements, where we work with a reference element via pull-back functions Φ. For the local shape function of edge elements we have following relation between the curl on an arbitrary triangle and the curl on the reference triangle as shown in (2.1).

$$\mathrm{curl}_{2\mathrm{D}} \boldsymbol{b} = \det D\Phi_K^{-1} \ \mathrm{curl}_{2\mathrm{D}} \hat{b}$$

We apply this to the integral of one element matrix entry and we get

$$\begin{aligned}
&\int_K \mu(x) \ \mathrm{curl}_{2\mathrm{D}} \boldsymbol{b}_i \cdot \ \mathrm{curl}_{2\mathrm{D}} \boldsymbol{b}_j \mathrm{d}\boldsymbol{x} = \\
&\int_{\hat{K}} \mu(\Phi(\hat{x}))[\det D\Phi^{-1} \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{x})] \cdot [\det D\Phi^{-1} \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{x})] |\det D\Phi| \mathrm{d}\boldsymbol{x} \approx \\
&\sum_{l=1}^{P} \hat{w}(\mu(\Phi(\hat{\xi}_l))[\det D\Phi^{-1} \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{\xi}_l)] \cdot [\det D\Phi^{-1} \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{\xi}_l)] |\det D\Phi|) = \\
&\sum_{l=1}^{P} \hat{w}(\mu(\Phi(\hat{\xi}_l)) \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{\xi}_l) \cdot \ \mathrm{curl}_{2\mathrm{D}} \hat{\boldsymbol{b}}_i(\hat{\xi}_l) |\det D\Phi|^{-1})
\end{aligned}$$

where $\hat{K}$ the reference element, $\hat{w}_i$ the weights on the reference element, $\hat{\xi}_l$ the quadrature nodes on the reference element and Φ the mapping from the reference element to the actual element.

Listing 4.1: Local assembler for curl curl with numerical quadrature

```
struct LocalCurl2dAssembler{
  private:
    static const int dim_ = 2; // world dimension (2D)
    using refEl_t = eth::base::RefElType;
    static const refEl_t REtria = refEl_t::TRIA;
  public:
    typedef double numeric_t;
    typedef Eigen::Matrix< numeric_t, 3, 3 > result_t;
    typedef Eigen::Matrix< numeric_t, 3, 1 > vector_t;
    typedef Eigen::Matrix< numeric_t, dim_,1> coord_t;
    typedef fe::FEBasis< fe::Linear, fe::FEBasisType::Curl > fe_basis_t;

    static void initialize() {}

    template< typename MU_T, typename ELEMENT >
    inline
    static result_t eval( const MU_T& mu, const ELEMENT& el )
    {
      ETH_ASSERT_MSG( el.refElType() == REtria ,
      "For this, integration only works with 2D triangles." );

      //Get element geometry and area
      const auto& geom = el.geometry();
      auto elem_area = geom.volume();
      //Select quadrature rule
      using quadtria_t = betl2::quad::Quadrature< REtria, 3>;
      //Get points and weights over reference triangle
      const auto & wti = quadtria_t::getWeights()*quadtria_t::getScale();
      const auto & xti = quadtria_t::getPoints();
      //Get determinant of Jacobian of 'reference->actual'
      //element transformation
      const auto detJi = geom.template integrationElement< 3 >( xti );
      //Map quadrature points and weights to current triangle
      //Here inverse of determinante due to transformation of
      //the curl of the basis function
      const auto  globwti = detJi.cwiseInverse().cwiseProduct( wti );
      const auto&  globxti = geom.global(xti);
      //Get curl of basis functions for reference triangle;
      typedef typename
        fe_basis_t::template diffBasisFunction_t< REtria> basisFuncts;
      //Evaluate them on quadrature points
      const auto functEval  = basisFuncts::Eval( xti );

      vector_t orientation;
      //Get local orientation of edge with respect to global orientation
      for(int i=0; i<3; i++)
        orientation(i)= el.template orientationSign<1>(i) ?  1 : -1;

      result_t result;
      result.setZero();

      //For every quadrature point
      for( int l=0; l < xti.cols(); l++ ){
        //Fetch basis functions evaluated at current quadrature point
```

```
55          //and apply orientation correction
56          const  auto curl_l =
57            functEval.template  block< 3, 1 >( 0, l ).cwiseProduct(orientation);
58          //Evaluate mu function at current quadrature point
59          const auto MUeval = mu( globxti.col(l) );
60
61          result += globwti(l) * curl_l * MUeval* curl_l.transpose() ;
62        }
63
64      return result;
65    }
66  };  //end class LocalCurl2dAssemvler
```

As mentioned in 2.1 the orientation of the edge is rather important for edge elements and has to be treated with care. In BETL the global orientation of the edges is stored in the elements of the finite element space itself. We can access orientation through the element and the local index of the edge with respect to this element.

4.1.2 Assembly of mixed matrix

The integral $\int_K \boldsymbol{a}\ \mathbf{grad}\ \varphi dx$ can be evaluated analytically. Here we have one local shape function of the edge element and one linear Lagrange local shape function. One entry of the element matrix is

$$\int_K \boldsymbol{W}_i \cdot\ \mathbf{grad}\ b_k \mathrm{d}\boldsymbol{x} =$$
$$d_K^i(\ \mathbf{grad}\ \lambda_j \cdot\ \mathbf{grad}\ \lambda_k \int_K \lambda_i \mathrm{d}\boldsymbol{x} -\ \mathbf{grad}\ \lambda_i \cdot\ \mathbf{grad}\ \lambda_k \int_K \lambda_i \mathrm{d}\boldsymbol{x}) =$$
$$\frac{|K|}{3} d_K^i(\ \mathbf{grad}\ \lambda_j \cdot\ \mathbf{grad}\ \lambda_k -\ \mathbf{grad}\ \lambda_i \cdot\ \mathbf{grad}\ \lambda_k)$$

where $d_K^i = \pm 1$ is the orientation of the edge $\boldsymbol{e}_i$.

Listing 4.2: Local assembler for mixed matrix B

```
1  struct AnalyticGradAssembler{
2    private:
3      static const int dim_ = 2; // world dimension (2D)
4    public:
5      typedef double numeric_t;
6      typedef Eigen::Matrix< numeric_t, 3, 3 > result_t;
7      typedef Eigen::Matrix< numeric_t, 3, 1 > vector_t;
8      typedef Eigen::Matrix< numeric_t, dim_,1> coord_t;
9
10     static void initialize() {}
11
12     template< typename BUILDERDATA_T, typename ELEMENT >
13     inline
14     static result_t eval( const BUILDERDATA_T& data, const ELEMENT& el )
15     {
16       ETH_ASSERT_MSG( el.refElType() == eth::base::RefElType::TRIA ,
17       "For this, integration only works with 2D triangles." );
18
19       // Get element geometry and area
20       const auto& geom = el.geometry();
21       auto elem_area = geom.volume();
22
23       vector_t orientation;
24       //Get  global orientation of edge
```

```
25        for(int i=0; i<3; i++)
26          orientation(i)= el.template orientationSign<1>(i) ?  1 : -1;
27
28        Eigen::Matrix<numeric_t, 3,3> x;
29        x.block<3,1>(0,0) = Eigen::Vector3d::Ones();
30        x.block<1,2>(0,1) = geom.mapCorner(0).transpose();
31        x.block<1,2>(1,1) = geom.mapCorner(1).transpose();
32        x.block<1,2>(2,1) = geom.mapCorner(2).transpose();
33
34        // compute gradients
35        Eigen::MatrixXd grads  = x.inverse().block<2,3>(1,0);
36
37        // compute helper matrix
38        result_t r2 = grads.transpose()*grads*elem_area/12;
39
40        //compute element matrix
41        result_t result;
42        result<<orientation(0)*(r2(1,0)-r2(0,0)),
43            orientation(0)*(r2(1,1)-r2(0,1)),
44            orientation(0)*(r2(1,2)-r2(0,2)),
45            orientation(1)*(r2(2,0)-r2(1,0)),
46            orientation(1)*(r2(2,1)-r2(1,1)),
47            orientation(1)*(r2(2,2)-r2(1,2)),
48            orientation(2)*(r2(0,0)-r2(2,0)),
49            orientation(2)*(r2(0,1)-r2(2,1)),
50            orientation(2)*(r2(0,2)-r2(2,2));
51
52        return result;
53      }
54  };  //end class AnalyticGradAssembler
```

4.1.3 Assembly of lumped mass matrix

Here we assemble the lumped mass matrix. The integral $\int \varphi\psi dx$ is evaluated with 2D trapezoidal quadrature, such that the element matrix and the Galerkin matrix are diagonal. For the local shape functions of linear Lagrange elements we have

$$\int_K \lambda_i \lambda_j \mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3} \sum_{l=1}^{3} \underbrace{\lambda_i(\boldsymbol{a}_l)}_{\delta_{il}} \underbrace{\lambda_j(\boldsymbol{a}_l)}_{\delta_{lj}} = \delta_{ij} \frac{|K|}{3}.$$

And therefore the local element matrix is

$$\begin{pmatrix} \frac{|K|}{3} & 0 & 0 \\ 0 & \frac{|K|}{3} & 0 \\ 0 & 0 & \frac{|K|}{3} \end{pmatrix}.$$

Listing 4.3: Global assembler for lumped mass matrix

```
1  template<class FESPACE_TEST_T>
2  typename LumpedMassAssembler::vector_t
3  LumpedMassAssembler::assembleVector(const FESPACE_TEST_T  &  fe_test )
4  {
5    vector_t  res = vector_t::Zero(fe_test.numDofs());
6    //For all elements
7    for(const auto& el : fe_test)
8    {
```

```
    //Get element area
    const auto value = el.geometry().volume()/3.;
    //Map element contribution to global matrix
    for(const auto idx : fe_test.indices(el))
      res(idx.global()) += value;
  }

  return res;
}
```

In the code the local contributions are directly mapped to the global diagonal matrix. For efficiency the diagonal matrix is stored as vector.

4.1.4 Assembly of right hand side

For the calculation of the source current contribution, we also use the 2D trapezoidal rule. An entry of the local right hand side therefore is

$$\varphi_i = \int_K \boldsymbol{j} \cdot \boldsymbol{b}_i \mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3} \sum_{l=1}^{3} \boldsymbol{j}(\boldsymbol{a}_l) \cdot \boldsymbol{b}_i(\boldsymbol{a}_l).$$

We arrive at the local right hand side

$$\boldsymbol{\varphi}_K \approx \frac{|K|}{3} \begin{pmatrix} d_K^1(\boldsymbol{j}(\boldsymbol{a}_1) \cdot \textbf{grad}\ \lambda_2 - \boldsymbol{j}(\boldsymbol{a}_2) \cdot \textbf{grad}\ \lambda_1) \\ d_K^2(\boldsymbol{j}(\boldsymbol{a}_2) \cdot \textbf{grad}\ \lambda_3 - \boldsymbol{j}(\boldsymbol{a}_3) \cdot \textbf{grad}\ \lambda_2) \\ d_K^3(\boldsymbol{j}(\boldsymbol{a}_3) \cdot \textbf{grad}\ \lambda_1 - \boldsymbol{j}(\boldsymbol{a}_1) \cdot \textbf{grad}\ \lambda_3) \end{pmatrix}$$

where $d_K^i = \pm 1$ is again the orientation of the edge e_i.

Listing 4.4: Local assembler for right hand side

```
struct RhsAssembler{
  private:
    static const int dim_ = 2;  // world dimension (2D)

  public:
    typedef double numeric_t;
    typedef Eigen::Matrix< numeric_t, 3, 1 > result_t;
    typedef Eigen::Matrix< numeric_t, 3, 1 > vector_t;
    typedef Eigen::Matrix< numeric_t, dim_,1> coord_t;

    static void initialize() {}

    template< typename BUILDERDATA_T, typename ELEMENT >
    inline
    static result_t eval( const BUILDERDATA_T& j, const ELEMENT& el )
    {
      ETH_ASSERT_MSG( el.refElType() == eth::base::RefElType::TRIA ,
      "For this, integration only works with 2D triangles." );

      //Get element geometry and area
      const auto& geom = el.geometry();
      auto elem_area = geom.volume();

      vector_t orientation;
      //Calculate  global orientation of edge
      for(int i=0; i<3; i++)
        orientation(i)= el.template orientationSign<1>(i)  ?  1 : -1;
```

```
28
29        result_t result;
30        result.setZero();
31
32        Eigen::Matrix<numeric_t, 3,3> x;
33        x.block<3,1>(0,0) = Eigen::Vector3d::Ones();
34        x.block<1,2>(0,1) = geom.mapCorner(0).transpose();
35        x.block<1,2>(1,1) = geom.mapCorner(1).transpose();
36        x.block<1,2>(2,1) = geom.mapCorner(2).transpose();
37
38        //Compute gradients
39        Eigen::MatrixXd grads  = x.inverse().block<2,3>(1,0);
40
41        //Here index i is the local index of the edge-dof,
42        //but we use vertex based quadrature then the vertices
43        //are i and (i+1)%3.
44        for (unsigned i=0;i<3;++i){
45          //Local contribution computed by "2D-trapezoidal
46          //rule inspired quadrature"
47          result(i) = orientation(i)*elem_area/3.0 *
48            (j(geom.mapCorner(i)).dot(grads.col((i+1)%3))-
49            j(geom.mapCorner((i+1)%3)).dot(grads.col(i)));
50        }
51
52        return result;
53      }
54  };  //end class RhsAssembler
```

4.2 HYPRE

HYPRE is a library containing various linear solver and especially multigrid methods.

4.2.1 Matrix formats

IJ Format

HYPRE uses its own sparse matrix format. It is called IJ-format, as reference to the numbering of matrix components. The format is row major and it consists of five parts. First an integer that stores the number of rows in the matrix. Then two arrays of size number of rows, where one stores the indices of the rows and the other the number of non-zeros in the row. And finally there are two arrays, whose length correspond to the number of non-zero entries in the matrix. The first array stores the column indices of the entries and the second stores the actual values.

Name	Type	Size	Description
nrow	int	1	# of rows.
ncols	int array	nrow	# of non-zeros in the i-th row
rows	int array	nrow	Row index
cols	int array	nnz	Column index
values	value_type array	nnz	Values

Table 4.1: Overview of the parts of the IJ matrix format

For vectors HYPRE has a similar format, where simply all column related information is dropped.

Name	Type	Size	Description
nrow	int	1	# of rows.
rows	int array	nrow	Row index
values	value_type array	nnz	Values

Table 4.2: Overview of the parts of the IJ vector format

Eigen CRS

Eigen also supports a row major sparse matrix format. The format is the compressed row storage (CRS) format.

Name	Type	Size	Description
row_pointer	int array	# of rows	Index of the start of i-th row in column_index
column_index	int array	nnz	Column index
values	value_type array	nnz	Values

Table 4.3: Overview of the parts of CRS format

Example

A short example how the different formats store a sparse matrix to demonstrate the concept.

$$A = \begin{pmatrix} 2 & 0 & 5 & 9 & 0 \\ 0 & 0 & 3 & 0 & 2 \\ 4 & 0 & 0 & 1 & 0 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 4 & 5 & 7 & 0 \end{pmatrix}$$

Eigen CRS format:

row_pointer	0	3	5	7	8

column_index	0	2	3	2	4	0	3	2	1	2	3
values	2	5	9	3	2	4	1	8	4	5	7

HYPRE IJ format:

nrows	5

ncols	3	2	2	1	3
rows	0	1	2	3	4

cols	0	2	3	2	4	0	3	2	1	2	3
values	2	5	9	3	2	4	1	8	4	5	7

Here is the code that copies an Eigen CRS matrix to a matrix in HYPRE IJ format.

Listing 4.5: Matrix mapping from Eigen to HYPRE

```
1 void EigenMat2HypreMat(const Eigen::SparseMatrix<double,Eigen::RowMajor>& A,
2   int& nrows,
3   std::vector<int>& ncols,
4   std::vector<int>& rows,
5   std::vector<int>& cols,
```

```
6     std::vector<double>& values)
7 {
8   int nnz = A.nonZeros();
9   nrows = A.rows();
10
11  //Allocate memory
12  ncols.resize(nrows);
13  rows.resize(nrows);
14  cols.resize(nnz);
15  values.resize(nnz);
16
17  //Fill row index and count column per row
18  for(int i=0; i<nrows; i++)
19  {
20    //Row index
21    rows[i] = i;
22    //Calculate non-zeros per row
23    if(i<nrows -1)
24      ncols[i] = A.outerIndexPtr()[i+1] - A.outerIndexPtr()[i];
25    else
26      ncols[i] = nnz- A.outerIndexPtr()[i];
27  }
28
29  //Fill column index and values
30  for(int j =0; j < nnz; j++)  {
31    //Column index
32    cols[j] = A.innerIndexPtr()[j];
33    //Value
34    values[j] = A.valuePtr()[j];
35  }
36 }
```

Chapter 5

Numerical Experiments

Various experiments are conducted either to validate the implementation or to study the behaviour of the convergence of the algebraic multigrid method. First we introduce the used model problems, then we have a short glance at the validation and finally we dive into the results of the AMG convergence.

5.1 Model Problems

5.1.1 Poisson

We use the Poisson equation

$$-\Delta u = f \text{ in } \Omega, u = 0 \text{ on } \partial\Omega \tag{5.1a}$$

with $\Omega = [0,1] \times [0,1]$.

As model problem we have:

$$u(x,y) = \sin(2\pi y)\sin(2\pi x) \tag{5.1b}$$

$$f(x,y) = 8\pi^2 \sin(2\pi y)\sin(2\pi x) \tag{5.1c}$$

This model problem is mainly used to validate the interface between Eigen and HYPRE and to get reference of the convergence of the AMG method.

5.1.2 Magnetostatics

We have two domains for the magnetostatic problem (1.1). One is on the square $\Omega = [0,1] \times [0,1]$ and the other is on the unit disk $D = \{x \colon \|x\| \leq 1\}$.

Square domain Here we assume we have following solution for (1.1).

$$\boldsymbol{a}(x,y) = \begin{pmatrix} \sin(\pi y) \\ \sin(\pi x) \end{pmatrix} \tag{5.2a}$$

Further we assume

$$\mu(x,y) = 1 + y. \tag{5.2b}$$

With $\boldsymbol{a}$ and μ we can construct the source that gives us the assumed $\boldsymbol{a}$ if we solve (1.1).

$$\boldsymbol{j}(x,y) = \begin{pmatrix} \pi(\cos(\pi x) - \cos(\pi y) + (1+y)\pi \sin(\pi y)) \\ \pi^2 (y+1)\sin(\pi x) \end{pmatrix} \tag{5.2c}$$

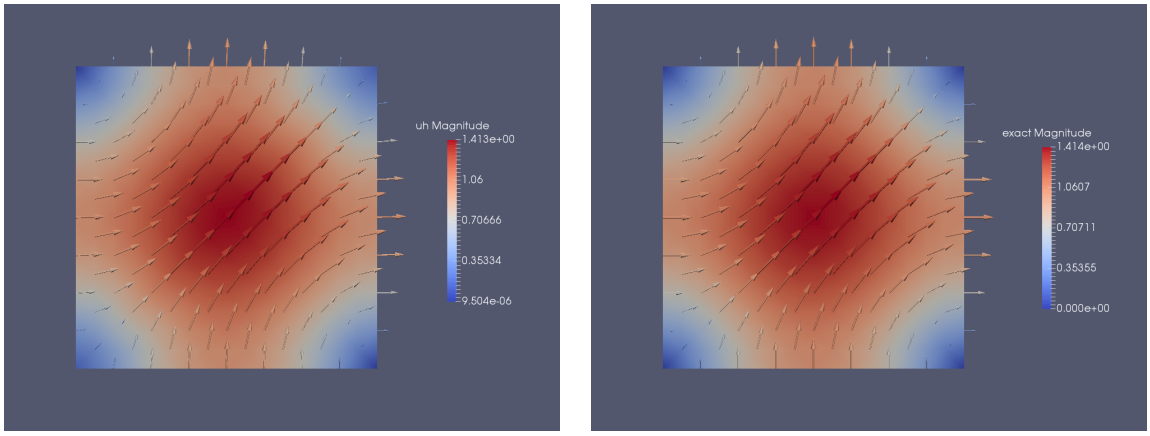

Figure 5.1: Numerical solution on the left and analytic solution on the right.

Circle domain For the model problem on the unit disk we assume a solution and μ. Then we construct the necessary source function from $\boldsymbol{a}$ and μ. We assume

$$\boldsymbol{a}(x,y) = \begin{pmatrix} \sin(\pi r)y \\ -\sin(\pi r)x \end{pmatrix} \tag{5.3a}$$

$$\mu(x,y) = 1 + x^2 \tag{5.3b}$$

where $r = x^2 + y^2$.

For the chosen $\boldsymbol{a}$ and μ we get the manufactured $\boldsymbol{j}$.

$$\boldsymbol{j}(x,y) = \begin{pmatrix} 4(x^2+1)y\pi(\pi r \sin(\pi r) - 2\cos(\pi r)) \\ 4x((x^2+1)\pi(2\cos(\pi r) - \pi r \sin(\pi r)) + \sin(\pi r) + \pi r \cos(\pi r)) \end{pmatrix} \tag{5.3c}$$

Since the boundaries on a circle cannot be exactly represented by a triangulation, we use the given linear approximation of the boundary by the triangulation.

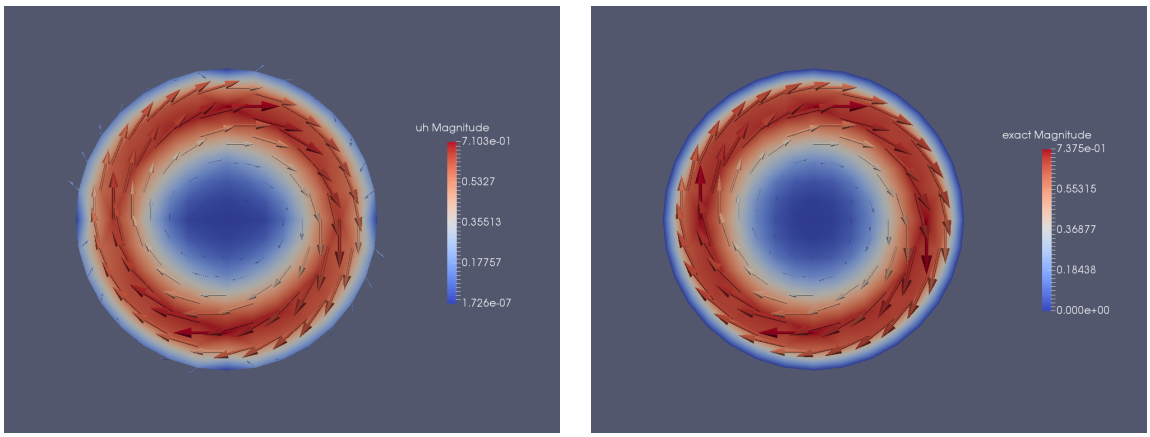

Figure 5.2: Numerical solution on the left and analytic solution on the right.

5.2 Validation

5.2.1 h-Convergence

Poisson From Theorem 5.3.38 in [3] we get the error estimates for piecewise linear interpolation.

$$\|u - l_1 u_n\|_{L^2(\Omega)} \leq O(h^2)$$
$$\|\,\mathbf{grad}\,(u - l_1 u_n)\|_{L^2(\Omega)} \leq O(h)$$

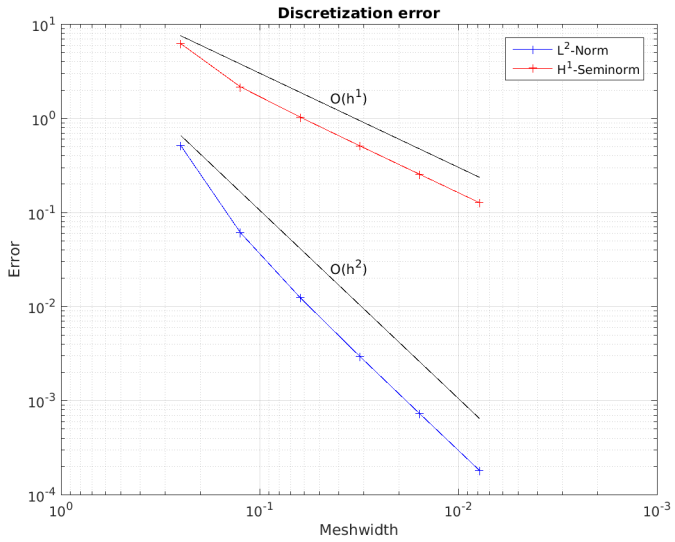

Figure 5.3: L^2-Norm and H^1- Seminorm of discretization error for (5.1) on $[0,1] \times [0,1]$

Magnetostatic From Theorem 3.14 in [2] we have the interpolation error estimate

$$\|\boldsymbol{u} - \Pi_1^1 \boldsymbol{u}\|_{L^2(\Omega)} \leq C h^{\min s, p+1} (\|u|_{\boldsymbol{H}^s(\Omega)} + \|\,\mathbf{curl}_{2\mathrm{D}}\,\boldsymbol{u}\|_{\boldsymbol{H}^s(\Omega)})$$

For the lowest order Whitney 1-forms we get the same rate of convergence in L^2 and $\|\,\mathbf{curl}_{2\mathrm{D}}\,\cdot\|_{L^2}$ norm see Remark 10, Chapter 3 in [2].

5.3 Convergence of AMG

In this section the results on the convergence of the AMG method are reported. For all experiments the HYPREBoomerAMG solver was used with Ruge coarsening and hybrid Gauss-Seidel relaxation. As stopping criterion a relative tolerance of 10^{-7} was used. Although HYPRE supports multiple cores through MPI, all experiments ran on single core.

5.3.1 Poisson model problem

The algebraic multigrid method was used to solve the Poisson model problem (5.1) on a series of refined meshes for $[0,1] \times [0,1]$. Two different refinement strategies have been used. First the mesh was regularly refined. These results are reported in table 5.1. The

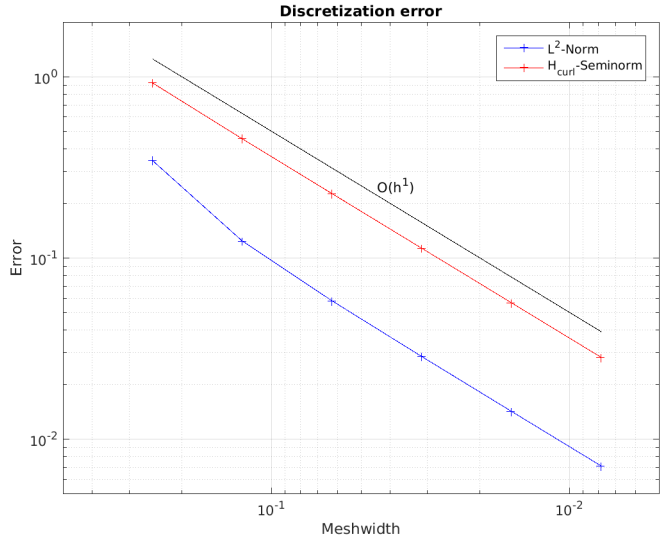

Figure 5.4: L^2-Norm and H_{curl}- Seminorm of discretization error for (5.2) on $[0,1] \times [0,1]$.

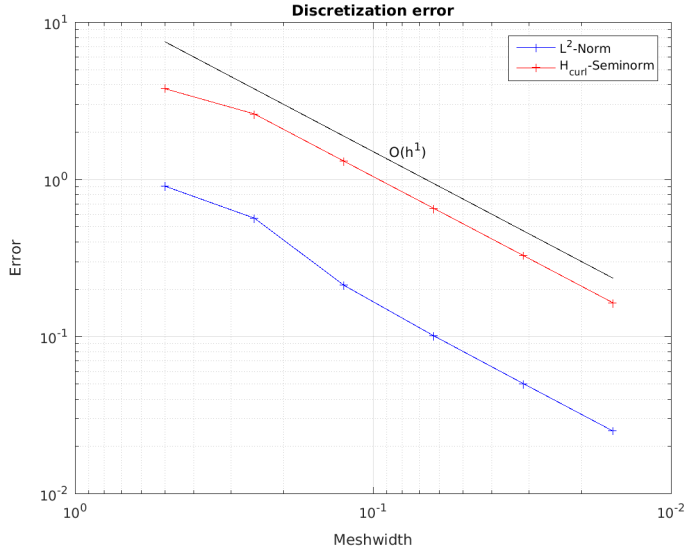

Figure 5.5: L^2-Norm and H^1-Seminorm of discretization error for (5.3) on unit disk.

second time the mesh was only locally refined along one edge. Those results are reported in table 5.2.

We see that the number of iterations is independent of the problem size and the number of iteration is relatively small. Also the structure of the mesh has no influence on the convergence rate of the algebraic multigrid method for the Poisson model problem.

5.3.2 Magnetostatics

The algebraic multigrid method was again used to solve the magnetostatics test cases. The solver was applied to a series of refined meshes. The results with regularly refined meshes for (5.2) on $[0,1] \times [0,1]$ are reported in table 5.3. In another experiment for the magnetostatic model problem (5.2) the mesh was refined along one edge. These results are reported

# of d.o.f.	# of iter	Final rel. residual
5	1	2.2497822192e-17
25	1	2.8518896223e-16
113	5	1.0273374838e-08
481	6	1.8888441723e-08
1985	7	8.2911430047e-09
8065	7	6.5158776181e-08

Table 5.1: Number of iteration and final residual on $[0,1] \times [0,1]$ with regular refinement for Poisson model (5.1)

# of d.o.f.	# of iter.	Final rel. residual
77	5	3.4801161165e-08
158	6	7.5348695323e-09
327	6	4.0351065942e-08
684	7	1.0760043373e-08
1340	7	2.1959227584e-08
2674	7	3.3400172126e-08

Table 5.2: Number of iteration and final residual on $[0,1] \times [0,1]$ with local refinement for Poisson model (5.1)

in table 5.5. For the second model problem (5.3) was once more the regular refinement applied. The results for regular refinement on the unit disk are reported in table 5.4. At last the circle domain was locally refined around one point in the inside. These results are shown in table 5.6.

We see that the number of iterations is no longer independent of the problem size. For regular refinement the number of iterations needed to attain the prescribed accuracy scales linear with the size of the system matrix. We also observe that the method of refinement has now an impact on the convergence. The local refinement slows the algebraic method significantly down. For the refinement along an edge the algebraic multigrid scales with $O(N^{\sqrt{2}})$.

# of d.o.f.	# of iter	Final rel. residual
20	163	9.8153606492e-08
88	513	9.9520480525e-08
368	1963	9.9515497720e-08
1504	8722	9.9867055235e-08
6080	37396	9.9963041827e-08
24448	155828	9.9996723951e-08

Table 5.3: Number of iteration and final residual for magnetostatic model problem (5.2) with regular refinement on $[0,1] \times [0,1]$

# of d.o.f.	# of iter	Final rel. residual
20	115	9.5497498704e-08
88	391	9.7562910330e-08
368	1652	9.9380902520e-08
1504	6977	9.9868175277e-08
6080	26965	9.9998623431e-08
24448	124508	9.9999092472e-08

Table 5.4: Number of iteration and final residual for magnetostatic model problem (5.3) with regular refinement on unit disk

# of d.o.f.	# of iter	Final rel. residual
260	2618	9.9996711732e-08
519	4598	9.9849542739e-08
1048	8345	9.9917660480e-08
2161	21266	9.9983706530e-08
4201	52851	9.9994224040e-08
8341	138984	9.9989624783e-08

Table 5.5: Number of iteration and final residual for magnetostatic model problem (5.2) with local refinement on $[0,1] \times [0,1]$

# of d.o.f.	# of iter	Final rel. residual
304	2265	9.9477656884e-08
403	2583	9.9752743542e-08
541	4076	9.9811891212e-08
625	5132	9.9853600286e-08
748	6080	9.9997406370e-08
817	9499	9.9840900186e-08

Table 5.6: Number of iteration and final residual for magnetostatic model problem (5.3) with local refinement on unit disk

5.4 Eigenvalues of system matrix

The eigenvalues of the system matrix have also been investigated. Due to the limitation of the direct eigenvalue solver in Eigen only the eigenvalues for small system matrices ($n \leq 2000$) have been computed.
We observe that the eigenvalues of the system matrix for the poisson model problem do not depend on the matrix size, table 5.7. For the eigenvalues of $\tilde{A}$ in (1.3) we have that the largest eigenvalue grows with the system size, table 5.9 and 5.8. Also if in $\tilde{A}$ a significant amount of eigenvalues is smaller than the rest, when we compare Figure 5.9 and 5.10 to Figure 5.8, where the eigenvalues for a system matrix of the Poisson model problem are plotted against their index. This might be an explanation why AMG doesn't perform as good for the magnetostatic problem as for the Poisson problem. The small error components related to the small eigenvalues cannot be removed by the multigrid approach, because they aren't well represented on the coarser grids.

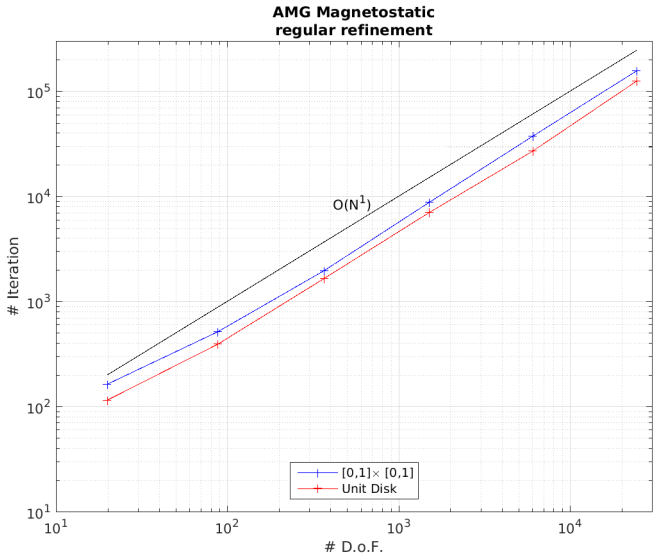

Figure 5.6: Iteration of AMG on regularly refined mesh for (5.2) (blue) and (5.3) (red).

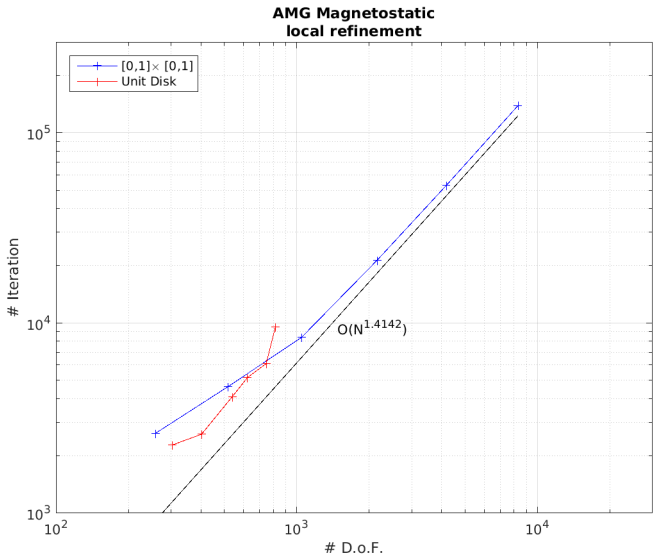

Figure 5.7: Iteration of AMG on locally refined mesh for (5.2) (blue) and (5.3) (red).

Size of matrix	λ_{min}	λ_{max}
5x5	2	6
25x25	0.585786437626906	7.41421356237311
113x113	0.152240934977427	7.84775906502259
481x481	0.038429439193539	7.96157056080639
1985x1985	0.009630546655587	7.99036945334438

Table 5.7: Smallest and largest eigenvalues of system matrix of different sizes for the Poisson model problem (5.1).

Size of matrix	λ_{min}	λ_{max}
20x20	0.9932194273302	141.4294096198
88x88	1.0317165288931	649.7535511774
368x368	1.0246909870621	2789.9555942352
1504x1504	1.0219938477535	11594.7616872224

Table 5.8: Smallest and largest eigenvalues of system matrix for different square meshes for the magnetostatic model problem (5.2).

Size of matrix	λ_{min}	λ_{max}
20x20	0.250561015504428	43.4669570577
88x88	0.237777556068641	188.5777410081
368x368	0.236336903071612	789.2730000082
1504x1504	0.236044204396962	3345.2717443891

Table 5.9: Smallest and largest eigenvalues of system matrix for different circle meshes for the magnetostatic model problem (5.3).

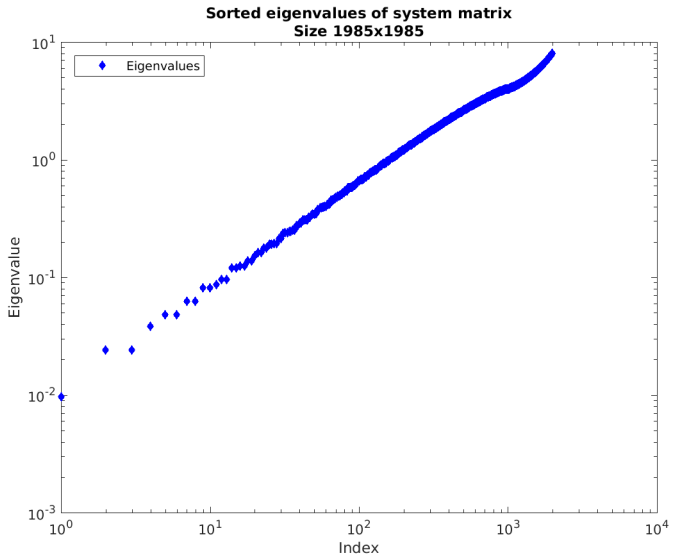

Figure 5.8: Sorted eigenvalues of system matrix for Poisson model problem (5.1) on $[0,1] \times [0,1]$.

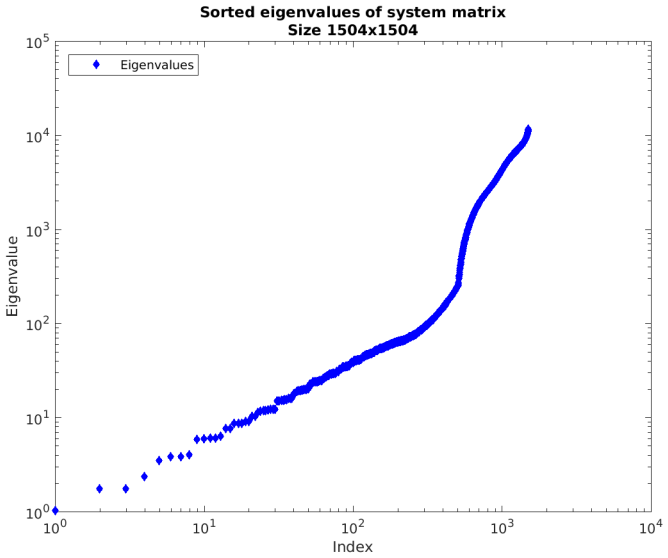

Figure 5.9: Sorted eigenvalues of system matrix $\tilde{A}$ for magnetostatic model problem (5.2) on $[0,1] \times [0,1]$.

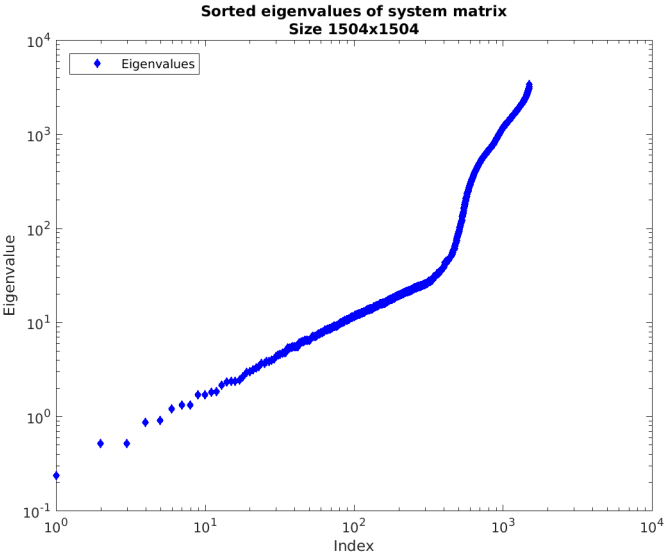

Figure 5.10: Sorted eigenvalues of system matrix $\tilde{A}$ for magnetostatic model problem (5.3) on unit disk.

Chapter 6

Conclusion

We used a Lagrange multiplier approach to get a variational formulation of the magnetostatic problem (1.1). This variational formulation was then discretized by lowest order edge elements and linear Lagrange elements. The assembly of the system matrix was implemented in the BETL framework. We also showed how we can convert an Eigen matrix to the HYPRE IJ matrix format. This conversion was used to prepare the matrix to be solved with the algebraic multigrid method implemented in HYPRE.

Numerical experiments have been conducted to investigate the behaviour of the 'classical' algebraic multigrid method when applied to the magnetostatic problem (1.1). The numerical experiments showed that the algebraic multigrid method is able to solve the Laplace problem in few iterations and there is no dependence on the problems size or mesh structure. But when applied to the magnetostatic problem the number of iterations becomes dependent on the problem size. It scales linearly with the problem size. When only a local refinement is applied, the number of iterations needed to achieve a prescribed accuracy is even larger.

While the algebraic multigrid method is efficient in solving Poisson type problems like (5.1), for magnetostatic type problems like (5.2) and (5.3), it seems not to be well suited. The reason for this could be the different distribution of the eigenvalues. The system matrices for the magnetostatic problem have a lot of small eigenvalues and a growing largest eigenvalues. But for the system matrix of the Poisson problem the eigenvalues are rather uniformly distributed on a restricted domain.

Appendix A

Appendix

A.1 Norms

Norms of the discritization error can be computed by means of numerical quadrature ([3], Remark 5.24).

A.1.1 L^2-Norm

$$\|u - u_N\|^2_{L^2(\Omega)} \approx \sum_{K \in M} \sum_{l=1}^{p} w_l \|(u - u_N)(\xi_l)\|^2$$

where $u_N = \sum_{i=1}^{N} \mu_i b_i$

Listing A.1: Struct to compute L^2-Norm

```
template<typename FESPACE_T, typename VECTOR_T,  typename FUNCTION_T>
static double L2Norm(
  FESPACE_T &fespace,
  const VECTOR_T &coeff,
  const FUNCTION_T& exact)
{

  ETH_ASSERT_MSG( el.refElType() ==  REtria,
  "For this, integration only works with 2D triangles." );

  // initialize result and set to zero
  double norm=0;

  //For all elements
  for(auto& el : fespace)
  {
    //Get element geometry and area
    const auto& geom = el.geometry();
    const auto indices = fespace.indices( el);
    //Select quadrature rule
    using quadtria_t = betl2::quad::Quadrature< REtria, 7>;
    //Get points and weights over reference triangle
    const auto & wti = quadtria_t::getWeights()*quadtria_t::getScale();
    const auto & xti = quadtria_t::getPoints();
    //Get determinant of Jacobian of 'reference->actual' element transformation
    const auto detJi = geom.template integrationElement< 7 >( xti );
    //Map quadrature points and weights to current triangle
    const auto  globwti = detJi.cwiseProduct( wti );
    const auto&  globxti = geom.global(xti);
```

```
30      //Get basis functions for reference triangle;
31      typedef typename
32        FESPACE_T::fe_basis_t::template basisFunction_t< REtria> basisFuncts;
33      //Evaluate them on quadrature points
34      const auto functEval  = basisFuncts::Eval( xti );
35      //Get jacobian matrix of affine transformation (inverted and transposed)
36      const Eigen::Matrix<double,2,14> JT =
37        el.geometry().template jacobianInverseTransposed<7>( xti );
38
39      //Get local coefficients
40      Eigen::Matrix<double,Eigen::Dynamic,1>
41        coeffLoc( FESPACE_T::fe_basis_t::numDofs(REtria) );
42      coeffLoc.setZero( );
43      for( const auto& idx : indices ) {
44        const int loc = idx.local();
45        const int glo = idx.global();
46        coeffLoc(loc) = coeff(glo);
47      }
48
49      //Impose linear combination
50      using imposeLC_t = fe::ImposeLinearCombination<FESPACE_T>;
51      const imposeLC_t imposeLC(fespace);
52      const auto imposedCoeffLoc = imposeLC(coeffLoc,el);
53
54      //For every quadrature point
55      for( int l=0; l < xti.cols(); l++ ){
56        Eigen::Matrix<double, 1, 1 > unum;
57        unum.setZero();
58        //Evaluate numerical solution at current quadrature point
59        for( const auto& idx : indices ) {
60          const int loc = idx.local();
61          unum += functEval.template block<1,1>( loc,l ) * imposedCoeffLoc(loc);
62        }
63        //Evaluate analytic function at current quadrature point
64        const auto uexact = exact( globxti.col(l) );
65        //And contribution to global integral
66        norm += globwti(l) * (uexact-unum).squaredNorm();
67      }
68    }
69
70    return std::sqrt(norm);
71  }
```

A.1.2 H^1-Seminorm

$$|u - u_N|^2_{H^1(\Omega)} \approx \sum_{K \in M} \sum_{l=1}^{p} \omega_l \| \textbf{grad}\ (u - u_N)(\xi_l)\|^2$$

where $u_N = \sum_{i=1}^{N} \mu_i b_i$

Listing A.2: Struct to compute H^1-Seminorm

```
1  template<typename FESPACE_T, typename VECTOR_T,  typename FUNCTION_T>
2  static double H1Norm(
3    FESPACE_T &fespace,
4    const VECTOR_T & coeff,
5    const FUNCTION_T& exact)
6  {
7      ETH_ASSERT_MSG( el.refElType() ==  REtria,
```

```
8     "For this, integration only works with 2D triangles." );
9
10    //Initialize result and set to zero
11    double norm=0;
12
13    //For all elements
14    for(auto& el : fespace)
15    {
16    //Get element geometry and area
17    const auto& geom = el.geometry();
18    const auto indices = fespace.indices( el);
19    //Select quadrature rule
20    using quadtria_t = betl2::quad::Quadrature< REtria, 7>;
21    //Get points and weights over reference triangle
22    const auto & wti = quadtria_t::getWeights()*quadtria_t::getScale();
23    const auto & xti = quadtria_t::getPoints();
24    //Get determinant of Jacobian of 'reference->actual' element transformation
25    const auto detJi = geom.template integrationElement< 7 >( xti );
26    //Map quadrature points and weights to current triangle
27    const auto  globwti = detJi.cwiseProduct( wti );
28    const auto&  globxti = geom.global(xti);
29    //Get gradient of basis functions for reference triangle;
30    typedef typename
31      FESPACE_T::fe_basis_t::template diffBasisFunction_t< REtria> basisFuncts;
32    //Evaluate them on quadrature points
33    const auto functEval  = basisFuncts::Eval( xti );
34    //Get jacobian matrix of affine transformation (inverted and transposed)
35    const Eigen::Matrix<double,2,14> JT =
36      el.geometry().template jacobianInverseTransposed<7>( xti );
37
38    //Get local coefficients
39    Eigen::Matrix<double,Eigen::Dynamic,1>
40      coeffLoc( FESPACE_T::fe_basis_t::numDofs(REtria) );
41    coeffLoc.setZero( );
42    for( const auto& idx : indices ) {
43      const int loc = idx.local();
44      const int glo = idx.global();
45      coeffLoc(loc) = coeff(glo);
46    }
47
48    //Impose linear combination
49    using imposeLC_t = fe::ImposeLinearCombination<FESPACE_T>;
50    const imposeLC_t imposeLC(fespace);
51    const auto imposedCoeffLoc = imposeLC(coeffLoc,el);
52
53    //For every quadrature point
54    for( int l=0; l < xti.cols(); l++ ){
55      Eigen::Matrix<double, 2, 1 > unum;
56      unum.setZero( );
57      //Evaluate numerical solution at current quadrature point
58      for( const auto& idx : indices ) {
59        const int loc = idx.local();
60        unum += functEval.template block<1,2>( loc, 2*l ).transpose() *
61          imposedCoeffLoc(loc);
62      }
63      //Evaluate analytic function at current quadrature point
64      const auto uexact = exact( globxti.col(l) );
65      //And contribution to global integral
66      norm += globwti(l) *
67        (uexact-JT.template block<2,2>(0,2*l)*unum).squaredNorm();
68    }
```

```
69    }
70
71      return std::sqrt(norm);
72  }
```

A.1.3 $H_{\mathbf{curl}}$-Seminorm

$$|\boldsymbol{u}-\boldsymbol{u}_N|^2_{H_{\mathbf{curl}}(\Omega)} = \| \operatorname{curl}_{2D}(\boldsymbol{u}-\boldsymbol{u}_N)(\xi_l)\|^2_{L^2(\Omega)} \approx \sum_{K\in M}\sum_{l=1}^{p}\omega_l \| \operatorname{curl}_{2D}(\boldsymbol{u}-\boldsymbol{u}_N)(\xi_l)\|^2$$

where $\boldsymbol{u}_N = \sum_{i=1}^{N}\mu_i \boldsymbol{b}_i$

Listing A.3: Struct to compute $H_{\mathbf{curl}_{2D}}$-Seminorm

```
1  template<typename FESPACE_T, typename VECTOR_T,  typename FUNCTION_T>
2  static double HCurlNorm(
3    FESPACE_T &fespace,
4    const VECTOR_T &coeff,
5    const FUNCTION_T& exact)
6  {
7    ETH_ASSERT_MSG( el.refElType() ==  REtria,
8    "For this, integration only works with 2D triangles." );
9
10   // initialize result and set to zero
11   double norm=0;
12
13   //For all elements
14   for(auto& el : fespace)
15   {
16     //Get element geometry and area
17     const auto& geom = el.geometry();
18     const auto indices = fespace.indices( el);
19     //Select quadrature rule
20     using quadtria_t = betl2::quad::Quadrature< REtria, 7>;
21     //Get points and weights over reference triangle
22     const auto & wti = quadtria_t::getWeights()*quadtria_t::getScale();
23     const auto & xti = quadtria_t::getPoints();
24     //Get determinant of Jacobian of 'reference->actual' element transformation
25     const auto detJi = geom.template integrationElement< 7 >( xti );
26     //Map quadrature points and weights to current triangle
27     const auto  globwti = detJi.cwiseProduct( wti );
28     const auto&  globxti = geom.global(xti);
29     //Get curl of basis functions for reference triangle;
30     typedef typename
31       FESPACE_T::fe_basis_t::template diffBasisFunction_t< REtria> basisFuncts;
32     //Evaluate them on quadrature points
33     const auto functEval  = basisFuncts::Eval( xti );
34     //Get jacobian matrix of affine transformation (inverted and transposed)
35     const Eigen::Matrix<double,2,14> JT =
36       el.geometry().template jacobianInverseTransposed<7>( xti );
37
38     //Get local coefficients
39     Eigen::Matrix<double,Eigen::Dynamic,1>
40       coeffLoc( FESPACE_T::fe_basis_t::numDofs(REtria) );
41     coeffLoc.setZero( );
42     for( const auto& idx : indices ) {
43     const int loc = idx.local();
44     const int glo = idx.global();
45     coeffLoc(loc) = coeff(glo);
```

```
46      }
47
48      //Impose linear combination
49      using imposeLC_t = fe::ImposeLinearCombination<FESPACE_T>;
50      const imposeLC_t imposeLC(fespace);
51      const auto imposedCoeffLoc = imposeLC(coeffLoc,el);
52
53      // For every quadrature point
54      for( int l=0; l < xti.cols(); l++ ){
55        Eigen::Matrix<double, 1, 1 > unum;
56        unum.setZero( );
57        //Evaluate numerical solution at current quadrature point
58        for( const auto& idx : indices ) {
59          const int loc = idx.local();
60          unum += functEval.template block<1,1>( loc,l ) * imposedCoeffLoc(loc);
61        }
62        //Evaluate analytic function at current quadrature point
63        const auto uexact = exact( globxti.col(l) );
64        //And contribution to global integral
65        norm += globwti(l) *
66          (uexact-JT.template block<2,2>(0,2*l).transpose().determinant()*unum)
67          .squaredNorm();
68      }
69    }
70
71    return std::sqrt(norm);
72  }
```

Appendix B

Appendix

B.1 .vtu Exporter for edge d.o.f in BETL

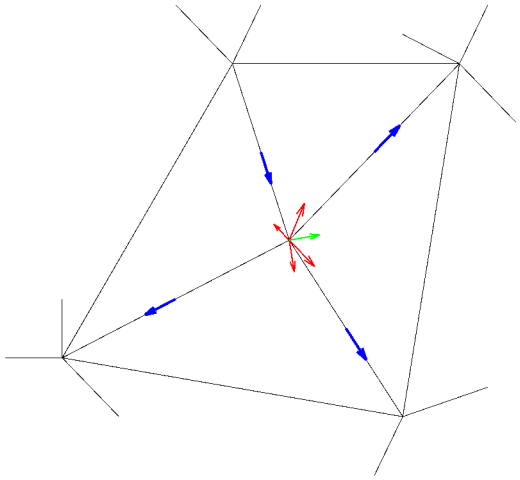

Figure B.1: The local contribution (red) from the adjacent triangles are average at the vertex (green).

Here are the code parts that where modified to support edge d.o.f interpolation in the triangle vertices.

Listing B.1: Modified .vtu Exporter for edge d.o.f. of BETL

```
enum class Entity : char { Point,  Cell, Edge};
/// interface for exporting grid functions
template< typename GRID_FUNCTION_T >
Exporter& operator()(
  const std::string descriptor,
  const GRID_FUNCTION_T& gridFunction,
  Entity entityType )
{
  typedef typename GRID_FUNCTION_T::gridFunctionTraits_t gridFunctionTraits_t;
  typedef typename gridFunctionTraits_t::numeric_t numeric_t;
  const int functionDim = gridFunctionTraits_t::functionDim;

  switch( entityType ) {
```

```
14      case Entity::Point: {
15        const auto pointData =
16          CreatePointData_<numeric_t,functionDim>()
17            ( gridFunction, grid_factory_, vertexLocalNumbering_ );
18        ExportData_<numeric_t,functionDim>()
19          ( pointOutputBuffer_, descriptor, pointData );
20        withPointData_ = true;
21      }
22      break;
23      case Entity::Cell:  {
24        const auto cellData =
25          CreateCellData_<numeric_t,functionDim>()
26            ( gridFunction, grid_factory_ );
27        ExportData_<numeric_t,functionDim>()
28          ( cellOutputBuffer_, descriptor, cellData );
29        withCellData_ = true;
30      }
31      break;
32      case Entity::Edge: {
33        const auto pointData =
34          CreateEdgeData_<numeric_t,functionDim>()
35            ( gridFunction, grid_factory_, vertexLocalNumbering_);
36        ExportData_<numeric_t,functionDim>()
37          ( pointOutputBuffer_, descriptor, pointData );
38        withPointData_ = true;
39      }
40      break;
41      default:
42        std::cerr << "vtu::Exporter::operator(): Unknown vtu-entity!" << std::endl;
43        exit( -1 );
44    }
45    return *this;
46  }
47  //Interpolates egde coeff to nodes
48  template< typename NUMERIC_T, int FUNCTION_DIM >
49  struct CreateEdgeData_
50  {
51    typedef ExpandVector_<FUNCTION_DIM> expandVector_t;
52    typedef Eigen::Matrix< NUMERIC_T, expandVector_t::dim, 1 > data_t;
53    typedef std::map< int, data_t > result_t;
54    typedef std::map<int, int> count_t;
55
56    template< typename GRID_FUNCTION_T >
57    result_t operator()(
58      const GRID_FUNCTION_T& gridFunction,
59      const GRID_FACTORY_T& grid_factory,
60      const local_point_numbering_t& vertexLocalNumbering ) const
61    {
62      const auto gridView = grid_factory.getView( );
63      const auto elements = gridView.template entities<0>( );
64      const node_mapper_t indexSet( gridView.indexSet( ) );
65
66      //Map that stores the values at the interpolation points
67      result_t dataset;
68      //Map that stores the number of elements that contirbuted to
69      //an interpolation point
70      count_t count;
71      //For all elements
72      for( const auto& E : elements ) {
73        //Get the reference type
74        const auto ret = E.refElType( );
```

```
75        //Get the number of nodes for this reference element
76        const int codimPoint  = dimFrom_;
77        const int numNodesRet =
78          eth::base::ReferenceElements::numSubEntities( ret, codimPoint );
79
80        typedef Eigen::Matrix<NUMERIC_T, 3, 1> vector_t;
81
82        //Traverse the nodes
83        for( int n = 0; n < numNodesRet; ++n ) {
84          const int pointID = indexSet.template subMap<dimFrom_>( E, n );
85          const int localPointID = vertexLocalNumbering.at( pointID );
86          const auto coords = eth::base::ReferenceElements::getNodeCoord( ret, n );
87          const auto coords_wrapper =
88            Eigen::Map<const Eigen::Matrix<double,dimFrom_,1>>( &coords[0] );
89          auto res = expandVector_t()(gridFunction(coords_wrapper,E));
90          res = res;
91
92          //Try to add contirbution to interpolation point
93          auto ret = dataset.insert( std::make_pair( localPointID, res ) );
94          //If ret.second == false, there was another element,
95          //that had a contribution to this intelerpolation point,
96          //we have to add the current value to the previous values
97          if(ret.second == false)
98          {
99            //Add value from current element at interoplation point
100           ret.first->second  += res;
101         }
102
103         auto c = count.insert(std::make_pair(localPointID, 1));
104         if(c.second == false) //Already a contribution to this point
105         {
106           //Increment the count
107           c.first->second += 1;
108         }
109       }
110     }
111
112     //For all interpolation points
113     for( auto& it : dataset)
114     {
115        //Average on point
116        it.second/=count.at(it.first);
117     }
118     return dataset;
119   }
120 }; // end struct CreatePointEdge_
```

Appendix C

Appendix

C.1 Manual

The complete source code can be found here: https://gitlab.ethz.ch/cmuenger/AMGRegMagStat

Installation and building

We can download and install the project with the following commands.

```
git clone --recursive https://gitlab.ethz.ch/cmuenger/AMGRegMagStat.git
cd AMGRegMagStat
mkdir build
cd build
module load mpi
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

More details about the installation of the project and its dependencies are available in the `README` of the repository. To be able to load the BETL submodule one must have access to the git repository of the NPDE16 lecture.

Excuting the examples

We can also build the model problems (5.1), (5.2) and (5.3) separately with the following commands.

```
% Model problem (5.2)
make AMG_hypre_magneto_square

% Model problem (5.3)
make AMG_hypre_magneto_circle

% Model problem (5.1)
make AMG_hypre_poisson
```

We can execute them from the `build` folder. We simply have to provide the path to the desired refined meshes.

```
% regular refinement
./bin/AMG_hypre_magneto_square ../meshes/regular/square
./bin/AMG_hypre_magneto_circle ../meshes/regular/circle
./bin/AMG_hypre_poisson ../meshes/regular/square

% local refinement
./bin/AMG_hypre_magneto_square ../ meshes/local/square_edge
./bin/AMG_hypre_magneto_circle ../ meshes/local/circle
./bin/AMG_hypre_poisson ../ meshes/local/square_edge
```

After a successful execution the information about the solving process is appended to `solver.txt`. The eigenvalues of the smaller system matrices are appended to `eigenvalues.txt`. And the behaviour of the discretization error is reported in `norm.txt`. The visualization of the solution is stored at the same location as the mesh.

Doxygen documentation

The doxygen documentation can be built in the `documentation` folder.

```
cd documentation
make doc
```

Bibliography

[1] PB Bochev and RB Lehoucq. Regularization and stabilization of discrete saddle-point variational problems. *Electronic Transactions on Numerical Analysis*, 22:97–113, 2006.

[2] R. Hiptmair. Finite elements in computational electromagnetism. *Acta Numerica*, 11:237–339, 2002.

[3] R. Hiptmair. Numerical methods for partial differential equations, 2016. Lecture Notes.

[4] Fumio Kikuchi. Mixed formulations for finite element analysis of magnetostatic and electrostatic problems. *Japan Journal of Applied Mathematics*, 6(2):209, 1989.

[5] J. W. Ruge and K. Stüben. *4. Algebraic Multigrid*, pages 73–130.

[6] A. Schneebeli. An $H(\mathbf{curl};\Omega)$-conforming FEM: Nédélec's elements of first type, 2003.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.