A Scalable Eddy-Current Solver for ANSYS

Master Thesis

of

Siebenmann Stefan

February 24, 2010

Supervisor:
Prof. Dr. R. Hiptmair

ETH Zurich
Computational Science and Engineering
Seminar of Applied Mathematics

Contents

1 Introduction

We are interested in an efficient numerical solution of the following problem stemming from Maxwell's equations:

$$\begin{aligned} \mathbf{curl}\ \alpha\ \mathbf{curl}\ \mathbf{u} + \beta\mathbf{u} &= \mathbf{f} \ \text{ in } \Omega \ \subset \mathbb{R}^3 \\ \mathbf{u} \times \mathbf{n} &= 0 \ \text{ on } \partial\Omega \end{aligned} \tag{1.1}$$

with coefficients $\alpha > 0$, $\beta >= 0$. An extension to complex-valued coefficients (for the equation in the frequency domain) is possible. A Galerkin finite element discretization based on edge elements yields the linear system of equations

$$\mathbf{A}\vec{x} = \vec{b} \tag{1.2}$$

The use of a direct solver is restricted to smaller problems. The only option for bigger problems are iterative solvers. Hiptmair and Xu introduced in [2] an auxiliary space preconditioner for the problem 1.1. The application of the preconditioner involves the solution of several Poisson problems, which can be done very efficiently with AMG (algebraic multigrid) methods.

In this thesis I will implement the AMS (Auxiliary space Maxwell Solver) preconditioner for the software package ANSYS and test its performance on a couple of problems.

2 Theoretical Background

2.1 Eddy Current Model

Time harmonic electromagnetic fields are governed by Maxwell's equations in the frequency domain:

$$\begin{aligned} \mathbf{curl}\ \mathbf{E} &= -i\omega\mu(\mathbf{x})\mathbf{H} && \text{in } \Omega \subset \mathbb{R}^3 \\ \mathbf{curl}\ \mathbf{H} &= (i\omega\varepsilon(\mathbf{x}) + \sigma(\mathbf{x}))\mathbf{E} + \mathbf{j}_o && \text{in } \Omega \subset \mathbb{R}^3 \end{aligned} \tag{2.1}$$

-$\mathbf{E}$: electric field
-$\mathbf{H}$: magnetic field
-μ: magnetic permeability
-ε: dielectric permittivity
-σ: conductivity
-$\mathbf{j}_0$: source current
-ω: angular frequency

For small ε we get the eddy current model:

$$\begin{aligned} \mathbf{curl}\ \mu^{-1}\ \mathbf{curl}\ \mathbf{E} + i\omega\sigma\mathbf{E} &= -i\omega\mathbf{j}_0 \text{ in } \Omega \\ \mathbf{E} \times \mathbf{n} &= 0 \qquad \text{on } \partial\Omega \end{aligned} \tag{2.2}$$

2.2 Variational Problem

We consider the following model boundary value problem

$$\begin{aligned} \mathbf{curl}\ \alpha(\mathbf{x})\ \mathbf{curl}\ \mathbf{u} + \beta(\mathbf{x})\mathbf{u} &= f \text{ in } \Omega \\ \mathbf{u} \times \mathbf{n} &= 0 \text{ on } \partial\Omega \end{aligned} \tag{2.3}$$

At first we will only deal with real valued coefficients α, β. Integration by parts leads then to the following variational problem:

$$\begin{aligned} &\text{seek } \ \mathbf{u} \in \mathbf{H}_0(\mathbf{curl}, \Omega), \text{ s.t. } \forall \mathbf{v} \in \mathbf{H}_0(\mathbf{curl}, \Omega): \\ &\int_\Omega \alpha(\mathbf{x})\,\mathbf{curl}\ \mathbf{u}\ \cdot \mathbf{curl}\ \mathbf{v} + \beta(\mathbf{x})\,\mathbf{u} \cdot \mathbf{v}\ \mathbf{dx} = \int_\Omega \mathbf{f} \cdot \mathbf{v}\mathbf{dx} \end{aligned} \tag{2.4}$$

Where the function space $\mathbf{H}_0(\mathbf{curl}, \Omega)$ is defined as:

$$\mathbf{H}_0(\mathbf{curl}, \Omega) := \left\{\mathbf{v} \in \mathbf{L}^2(\Omega) : \mathbf{curl}\ \mathbf{v} \in \mathbf{L}^2(\Omega), \mathbf{v} \times \mathbf{n} = 0 \text{ on } \partial\Omega\right\} \tag{2.5}$$

2.3 Finite Element Galerkin Discretization

Replacing $\mathbf{H}_0(\mathbf{curl}, \Omega)$ with the finite element space $E_h(M) \subset \mathbf{H}_0(\mathbf{curl}, \Omega)$ yields the discrete variational problem

$$\begin{aligned} &\text{seek } \ \mathbf{u} \in E_h(M), \text{ s.t. } \forall \mathbf{v} \in E_h(M): \\ &a(\mathbf{u}, \mathbf{v}) := (\alpha(\mathbf{x})\,\mathbf{curl}\ \mathbf{u},\ \mathbf{curl}\ \mathbf{v})_\Omega + (\beta(\mathbf{x})\,\mathbf{u}, \mathbf{v})_\Omega = (\mathbf{f}, \mathbf{v})_\Omega \end{aligned} \tag{2.6}$$

Here $E_h(M)$ denotes the edge finite element space on the tetrahedral mesh M. E_h is defined as:

$$E_h(M) = \left\{ v_h \in \mathbf{H}_0(\mathbf{curl}, \Omega) \;\; \mathbf{u}_h(\mathbf{x})|_K = \mathbf{a} + \mathbf{b} \times \mathbf{x}, \quad \mathbf{a}, \mathbf{b} \in \mathbb{R}^3, \; K \in M \right\} \tag{2.7}$$

The space E_h has a basis of locally supported functions associated with the edges of M with the property

$$\int_{\widehat{e}} \mathbf{b}_e \cdot d\mathbf{s} = \delta_{e,\widehat{e}} \tag{2.8}$$

The local shape functions of a tetrahedral element the can be expressed as:

$$\widetilde{\mathbf{b}}_{(i,j)} = \mathbf{grad}\lambda_i \cdot \lambda_j - \mathbf{grad}\lambda_j \cdot \lambda_i \tag{2.9}$$

where λ_i denotes the barycentric coordinate of node i.
Vector functions can be approximated in E_h by the use of the edge interpolation operator:

$$\mathrm{I}: \; \left(C^0(\Omega)\right)^3 \mapsto E_h, \quad \mathbf{u} \mapsto \sum_e \left(\int_e \mathbf{u} \cdot d\mathbf{s} \right) \mathbf{b}_e \tag{2.10}$$

Evaluation the variational problem 2.6 for pairs of basis functions leads to the linear system of equations

$$\mathbf{A}\vec{x} = \vec{b} \tag{2.11}$$

The Galerkin matrix $\mathbf{A}$ is symmetric positive (semi-)[1] definit and ill-conditioned. To solve the system efficiently with a conjugate gradient method one needs a good preconditioner. A preconditioner $\mathbf{B}$ should be an approximation of $\mathbf{A}^{-1}$ and its action on a vector $\vec{r}$ should be efficient to compute.

2.4 Auxiliary Space Preconditioning

A theoretical framework for finding a good preconditioner for the variational problem

$$u \in V: \; a(u,v) = f(u) \quad \forall v \in V, \; a(\cdot,\cdot) \text{ s.p.d.} \tag{2.12}$$

in the space V is the auxiliary space technique [5]. Its main building blocks are:

- auxiliary space W (finite element space) with inner product $d(\cdot,\cdot)$ on W
- continuous transfer operator $\Pi : W \mapsto V : \|\Pi w\|_A \leq c_\pi \|w\|_D$
- smoother $S : V \mapsto V'$ induced by a s.p.d. bilinear form $s(\cdot,\cdot)$ on V

The Auxiliary space preconditioner then takes the form:

$$B = S^{-1} + \Pi \circ D^{-1} \circ \Pi^* \tag{2.13}$$

The quality of the preconditioner measured by the condition number of the preconditioned system depends on the stability of the following splitting:

$$\exists c_0 > 0 : s(v_0, v_0) + d(w, w) \leq c_0 \|v\|_A^2, \quad \forall v = v_o + \Pi w \in V \tag{2.14}$$

$$\Rightarrow \mathrm{cond}(\mathbf{BA}) \leq c_0 \left(c_\pi^2 + \left\| S^{-1} \right\|^2 \right) \tag{2.15}$$

[1] if $\beta = 0$ for some materials in the domain

2.4.1 Discrete Stable Splittings

A starting point for finding a stable splitting for the finite element space E_h is the Helmholtz decomposition:

$$\begin{aligned}&\mathbf{u} = \mathbf{w} + \mathbf{grad}\ \phi,\ \text{div}\ \mathbf{w} = 0,\\ &\text{with}\ \mathbf{u} \in \mathbf{H}_0(\mathbf{curl};\Omega),\ \mathbf{w} \in \left(H_0^1(\Omega)\right)^3,\ \phi \in \mathbf{grad} H_0^1(\Omega)\end{aligned}$$

Indeed the Helmholtz decomposition offers a stable splitting for the space $\mathbf{H}_0(\mathbf{curl};\Omega)$ [2]. But obviously we are not interested in a preconditioner for $\mathbf{H}_0(\mathbf{curl};\Omega)$. The straightforward way to discretize the above decomposition is to replace $H_0^1(\Omega)$ by the space of continuous piecewise linear functions $S_{0,h}(M)$. Unfortunately $(S_{0,h})^3 \subset E_h$ does not hold in general, which requires the use of the edge interpolation operator. Furthermore $E_h \neq \mathbf{I}(S_{0,h})^3 + \mathbf{grad}\ S_{0,h}$ due to local high-frequency edge element functions, to capture these we need the smoother. This yields the following decomposition:
For all $\mathbf{v}_h \in E_h$ there exist $\boldsymbol{\psi}_h \in (S_{0,h}(\Omega))^3, \varphi_h \in S_{0,h}, \widetilde{\mathbf{v}}_h \in E_h$ such that

$$\mathbf{v}_h = \mathbf{I}\boldsymbol{\psi}_h + \widetilde{\mathbf{v}}_h + \mathbf{grad}\varphi_h \tag{2.16}$$

$$\|\boldsymbol{\psi}_h\|_{H^1(\Omega)} + \|\varphi_h\|_{H^1(\Omega)} + \left\|h^{-1}\widetilde{\mathbf{v}}_h\right\|_{L^2(\Omega)} \lesssim \|\mathbf{v}_h\|_{\mathbf{H}(\mathbf{curl};\Omega)} \tag{2.17}$$

This leads to the auxiliary space:

$$W := (S_{0,h})^3 \times S_{0,h} \tag{2.18}$$

with the transfer operator:

$$\Pi := \mathbf{I}_h \times \mathbf{grad} \tag{2.19}$$

2.4.2 Algorithm

The matrix representation of the auxiliary space preconditioner reads as follows

$$\mathbf{B} = \mathbf{S}_A^{-1} + \mathbf{P}\boldsymbol{\Delta}_h^{-1}\mathbf{P}^T + \mathbf{G}\Delta_h^{-1}\mathbf{G}^T \tag{2.20}$$

- $\mathbf{G}$ is the discrete gradient matrix, i.e. it represents the mapping : $\phi_h \in S_{0,h} \mapsto \mathbf{grad}\phi_h \in E_h$. It has the entries

$$G_{e,p} = \begin{cases} 0 & \text{, if } p \text{ is not an endpoint of } e \\ -1 & \text{, if } p \text{ is endpoint 1 of } e \\ 1 & \text{, if } p \text{ is endpoint 2 of } e \end{cases} \tag{2.21}$$

- The matrix $\mathbf{P}$ is the discrete version of the interpolation $\mathbf{I}_h : (S_{0,h})^3 \mapsto E_h$. If $\boldsymbol{\phi}_h \in (S_{0,h})^3$ then

$$\int_{e=(\mathbf{p}_1,\mathbf{p}_2)} \boldsymbol{\phi}_h \cdot d\mathbf{s} = \frac{\boldsymbol{\phi}_h(\mathbf{p}_2) - \boldsymbol{\phi}_h(\mathbf{p}_1)}{2} \cdot (\mathbf{p}_2 - \mathbf{p}_1) \tag{2.22}$$

For a coefficient vector $\vec{c}$ for $(S_{0,h})^3$ with three components c_p^x, c_p^y, c_p^z for every node $\mathbf{p}$ we therefore get (with $(e_x, e_y, e_z) := (\mathbf{p}_2 - \mathbf{p}_1)$)

$$(\mathbf{P}\vec{c})_e = \sum_{p \in e} \frac{1}{2} \left(e_x c_p^x + e_y c_p^y + e_z c_p^z \right) \tag{2.23}$$

- Δ is the Galerkin matrix of the bilinear form $(u, v) \mapsto \int_\Omega \beta \ \mathbf{grad}\ u \cdot \mathbf{grad}\ v \ d\mathbf{x}$ discretized on $S_{0,h}$ with standard node associated basis functions. The bilinear form corresponds to the Poisson problem $-\mathrm{div}\,(\beta\ \mathbf{grad}\cdot)$.
- $\mathbf{\Delta}$ is the Galerkin matrix of the bilinear form $(\mathbf{u},\ \mathbf{v}) \mapsto \int_\Omega \alpha\ \mathbf{grad}\ \mathbf{u} \cdot \mathbf{grad}\ \mathbf{v}\ d\mathbf{x}$ discretized on $(S_{0,h})^3$ with standard node associated basis functions. The bilinear form corresponds to the Poisson problem $-\mathrm{div}\,(\alpha\ \mathbf{grad}\cdot)$.
- $\mathbf{S}_A^{-1}$ is a symmetric Gauss-Seidel smoother

Note that $\mathbf{P}$ and $\mathbf{G}$ are local operators, they transfer values from the endpoints of each edge to the edge itself. $\mathbf{P}^T$, $\mathbf{G}^T$ transfer values from the edges to its endpoints with the same weights. Evaluation of the preconditioner $\mathbf{B}$ for a

vector $\vec{r}$ produces a result $\vec{c} = \mathbf{B}\vec{r}$ through the following steps:

1. Set $\vec{c} = \vec{0}$ and apply a symmetric Gauss-Seidel sweep to the system $\mathbf{A}\vec{c} = \vec{r}$
2. Set $\vec{\zeta} := \mathbf{G}^T\vec{r}$
3. Set $\vec{\rho} := \mathbf{P}^T\vec{r}$
4. Approximately solve $\mathbf{\Delta}\vec{\gamma} = \vec{\rho}$
5. Approximately solve $\Delta\vec{\kappa} = \vec{\zeta}$
6. Transfer corrections $\vec{c} \leftarrow \vec{c} + \mathbf{P}\vec{\gamma} + \mathbf{G}\vec{\kappa}$

An alternative form of the preconditioner is the multiplicative variant:

1. Set $\vec{c} = \vec{0}$ and apply a symmetric Gauss-Seidel sweep to the system $\mathbf{A}\vec{c} = \vec{r}$
2. Set $\vec{\zeta} := \mathbf{G}^T\left(\vec{r} - \mathbf{A}\vec{c}\right)$
3. Approximately solve $\Delta\vec{\kappa} = \vec{\zeta}$
4. Transfer correction $\vec{c} \leftarrow \vec{c} + \mathbf{G}\vec{\kappa}$
5. Set $\vec{\rho} := \mathbf{P}^T\vec{r}$
6. Approximately solve $\mathbf{\Delta}\vec{\gamma} = \vec{\rho}$
7. Transfer correction $\vec{c} \leftarrow \vec{c} + \mathbf{P}\vec{\gamma}$

8. Set $\vec{\zeta} := \mathbf{G}^T (\vec{r} - \mathbf{A}\vec{c})$
9. Approximately solve $\Delta\vec{\kappa} = \vec{\zeta}$
10. Transfer correction $\vec{c} \leftarrow \vec{c} + \mathbf{G}\vec{\kappa}$
11. Apply a symmetric Gauss-Seidel sweep to the system $\mathbf{A}\vec{c} = \vec{r}$

Some remarks:

- All of the above steps - except the steps involving the solution of the Poisson problems in the nodal spaces - have optimal computational complexity $\mathcal{O}(\dim E_h)$.
- The exact solution of these problems can be replaced by efficient AMG preconditioner, yielding an overall optimal computational effort.
- The matrix $\mathbf{\Delta}$ can be replaced by $\mathbf{P}^T\mathbf{A}\mathbf{P}$, the matrix Δ by $\mathbf{G}^T\mathbf{A}\mathbf{G}$

2.5 Complex Case

So far we only considered the case with real valued α, β. For the equations in the frequency domain β becomes imaginary. When discretizing equation 2.6 with complex coefficients one arrives at a complex valued linear system of equations

$$\vec{z} \in \mathbb{C}^n : \ \mathbf{A}\vec{z} = \vec{b} \quad \mathbf{A} \in \mathbb{C}^{n,n}, \ \vec{b} \in \mathbb{C}^n \tag{2.24}$$

This system of equations can be replaced by an equivalent one in $\mathbb{R}$

$$\underbrace{\begin{pmatrix} \mathbf{A}_R & -\mathbf{A}_I \\ -\mathbf{A}_I & -\mathbf{A}_R \end{pmatrix}}_{\mathbb{A}} \begin{pmatrix} \vec{z}_R \\ \vec{z}_I \end{pmatrix} = \begin{pmatrix} \vec{b}_R \\ -\vec{b}_I \end{pmatrix} \tag{2.25}$$

with

$$\begin{aligned} \mathbf{A} = \mathbf{A}_R + i\mathbf{A}_I \quad & \mathbf{A}_R, \mathbf{A}_I \in \mathbb{R}^{n,n} \\ \vec{b} = \vec{b}_R + i\vec{b}_I \quad & \vec{b}_R, \vec{b}_I \in \mathbb{R}^n \\ \vec{z} = \vec{z}_R + i\vec{z}_I \quad & \vec{z}_R, \vec{z}_I \in \mathbb{R}^n \end{aligned}$$

The matrix $\mathbb{A}$ is symmetric but not positive definite. Hence we use a (symmetric) MINRES instead of the conjugate gradient method to solve the system. To speed up the convergence we use the block preconditioner

$$\mathbb{B} = \begin{pmatrix} \mathbf{B} & \\ & \mathbf{B} \end{pmatrix} \tag{2.26}$$

where $\mathbf{B}$ is the AMS preconditioner for $\mathbf{A} = \mathbf{A}_R + \mathbf{A}_I$. A justification for this preconditioner can be found in A.1.

3 Implementation

3.1 Overview, Workflow

Preparation in ANSYS
The user creates a suitable electromagnetic problem with the use of element SOLID236 (for further information contact Dr. Henrik Nordborg[2]). The discretization of the model yields the Galerkin matrix $\mathbf{A}$ and the rhs $\vec{b}$. Mesh, physical data, Galerkin matrix and rhs will then be saved in the following ASCII files:

- testcase.cdb: physical model, mesh
- testcase.matrix: Galerkin matrix and rhs of the discretized model
- testcase.mapping: equation $\leftrightarrow$ edge(/node)

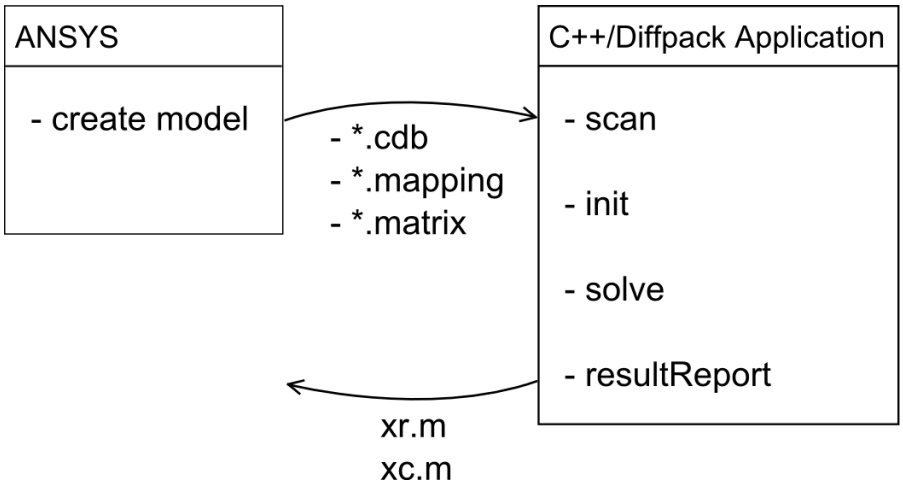


Figure 1: workflow

Solve with C++/Diffpack
The focus of this thesis is on the programming of a C++-application, which reads this data in and solves the linear system using a Krylov solver with a AMS preconditioner. The required mathematical functionality is provided by Diffpack. Diffpack is a numerical C++-library with main emphasis on numerical solution of partial differential equations. An online documentation can be found on `http://www.diffpack.com/`.

[2]CADFEM (Suisse) AG, Henrik Nordborg, henrik.nordborg@cadfem.ch

3.2 Discretization by ANSYS

ANSYS solves the Maxwell's equation by introducing potentials such that:

$$\mathbf{E} = -i\omega\mathbf{A} - \mathbf{grad}V \tag{3.1}$$

-$\mathbf{A}$: magnetic vector potential
-V: electric scalar potential

To ensure uniqueness of the discretized solution ANSYS uses a tree gauging algorithm that sets some degrees of freedom to zero. For the AMS preconditioner we have to disable the tree gauging algorithm and set instead all dofs associated with V to zero. To resulting variational formulation then becomes

$$\mathbf{u} \in E_h : \left(\mu^{-1}\mathbf{curl\ u}\ , \mathbf{curl\ v}\right) + i\omega\left(\sigma\mathbf{u}\ , \mathbf{v}\right) = \left(\mathbf{j}_0, \mathbf{v}\right) \quad \forall \mathbf{v} \in E_h \tag{3.2}$$

The values of $\mu = \mu_0\mu_{rx}$ and $\omega = 2\pi f$ are defined by the following parameters:

- EMUNIT for μ_0
- MURX for μ_{rx}
- RSVX for $\frac{1}{\sigma}$
- HARFERQ for f

3.3 DriverAndData

The main class for this project is called DriverAndData. As the name suggests, the class contains all the data (matrix, rhs of the system, grid, etc) that defines the problem. Furthermore the class provides functions to perform the steps needed to solve the linear system problem. If we look at the main function (in main.cpp):

```
int main (int argc, const char* argv[])
{

    initDiffpack (argc, argv);    // every Diffpack application
        starts with that command

    DriverAndData    driver;

    driver.scan();
    driver.init();
    driver.solveProblem();
    driver.resultReport();

}
```

We see four function calls corresponding to the steps in figure 1:

scan
The scan function creates an ANSYSInput object. The object then reads the text files with the problem data created by ANSYS and fills the related attributes of the Driver class:

- A_real: Sparse matrix (cf. A.2) representing the real part of the Galerkin matrix. Corresponds to the bilinear form $\left(\mu^{-1}\mathbf{curl}\ \mathbf{u}, \mathbf{curl}\ \mathbf{v}\right)$ on E_h.
- A_imag: Sparse matrix (cf. A.2) representing the imaginary part of the Galerkin matrix. Corresponds to the bilinear form $\omega\left(\sigma\mathbf{u},\ \mathbf{v}\right)$ on E_h.
- b_real: Real part of the rhs of the linear system
- b_imag: Imaginary part of the rhs of the linear system
- grid: the mesh created by ANSYS (cf. A.4)
- phys_data: simple structure containing the parameters of the model, as described in chapter 3.2. murx and rsvx are vectors, entry i corresponds to material i.
- edge_dofs: mapping equation $\leftrightarrow$ edge (cf. A.5)

The internals of the ANSYSInput class are described in chapter 3.4. To understand the implementation of the preconditioner one can skip that chapter and just remember the mentioned data structures describing the model.

init
The init function creates the linear system $\mathbf{A}\vec{x} = \vec{b}$ with

$$\mathbf{A} = \mathbf{A}_R + \mathbf{A}_I,\ \vec{b} = \vec{b}_R + \vec{b}_I \quad \text{real case} \tag{3.3}$$

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_R & -\mathbf{A}_I \\ -\mathbf{A}_I & -\mathbf{A}_R \end{pmatrix},\ \vec{b} = \begin{pmatrix} \vec{b}_R \\ -\vec{b}_I \end{pmatrix} \quad \text{complex case} \tag{3.4}$$

Secondly it computes alpha and beta from the phys_data structure.

$$\begin{aligned} \alpha(i) &= \frac{1}{emunit \cdot murx(i)} \\ \beta(i) &= \frac{2\pi \cdot harfreq}{rsvx(i)} \end{aligned} \tag{3.5}$$

Furthermore it sets the parameters for the Krylov solver:

```
if (complex){
    solv_prm->basic_method = "SymMinRes";
    conv_prm->residual_tp = LEFTPREC_RES;
}else{
    solv_prm->basic_method = "ConjGrad";
    conv_prm->residual_tp = ORIGINAL_RES;
}
```

```
solv_prm->startmode = USER_START;
solv_prm->max_iterations = 100;

conv_prm->monitor_tp = "CMRelResidual";       // |r_k|/|r_o|
    < tol
conv_prm->conv_tolerance = 1.0e-6;
conv_prm->norm_tp = l2;                       // 2-norm
```

Based on these choices it then creates a solver object. Finally it builds the AMS preconditioner, calls its init function and attaches the preconditioner to the linear system (cf. A.3).

solveProblem
Calls the solve function of the Krylov solver with the linear system as argument.

resultReport
Writes the solution to the hard disc (files: xr, xc) and gives some additional information about the convergence behavior.

3.4 ANSYS Input Reader

The ANSYSInput class reads the information given by the *.cdb, *.mapping and *.matrix file. The class uses the C++ stream classes fstream and stringstream. The class will first read a line from the file. If the line is relevant it will fill an istringstream object with the line. With the » operator we can then extract information from the istringstream object.
A simple example illustrates that procedure:
If we assume the first line from the *.mapping file is:

```
1    234   AZ
```

the following code will then fill the variables eqn, node and dof with the values 1, 234 and "AZ";

```
ifstream fin;
fin.open(dof_file.c_str());

string dof;
int eqn, node;

istringstream sline;
char line[80];

while (fin.getline(line, 80)){

    sline.clear();
    sline.str(line);

    sline >> eqn >> node >> dof;
```

...

}

3.4.1 cdb File

To allocate the necessary memory to save the grid data structures (cf. function redim of the GridFE class) we have to determine the numbers of elements, nodes, edges, materials etc. This is done by the function ReadMeshInfo. For the most values the read-out from the cdb-file is pretty simple. For example: For the number of elements one can find the line

NUMOFF, ELEM, 18456

in the cdb-file). The only difficulty is caused by the nodes and edges. ANSYS has no specific data format for edges, they are just represented by nodes located in the middle of the edge.

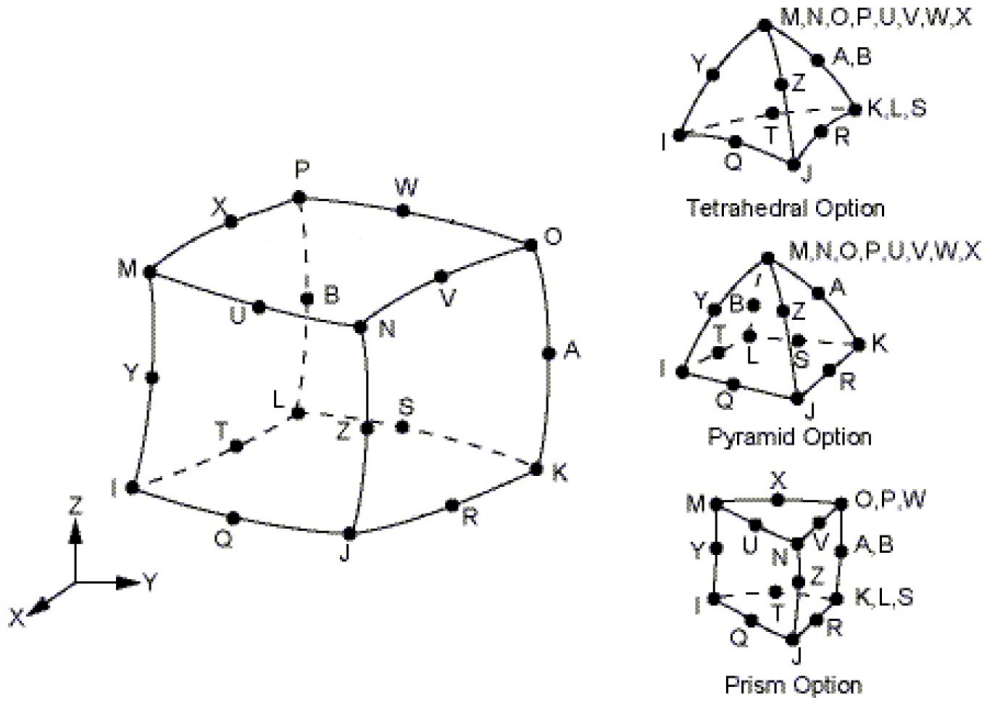


Figure 2: ANSYS: local nodes

For the purpose of the AMS preconditioner we are not interested in the coordinates of the midnode, instead we want to know for every edge which are his two endnodes. This requires that we save the edges and the (real) nodes in a different format (cf. A.4). The function labelEdges goes through the element definitions and labels all mid-nodes as edges in the array ANSYSnode_is_edge. Furthermore it creates an array (or more precisely a Diffpack vector) ANSYS_2_diffpack

that realizes the mapping: ANSYS node number $\mapsto$ Diffpack edge/node number. The renumbering is done in a way that preserves the property that the edge orientation is always from the lower (global) node number to the higher node number. With the information in these two arrays we can then determine the number of edges and nodes, allocate the necessary memory and fill all the grid data structures mentioned in A.4.

Edges and Elements
The elements are defined after the EBLOCK label. The cdb-file uses two lines for every element. The format of the first line is as follows:

- Field 1: material number of the element
- Field 2-11: - [3]
- Field 12-19: the global numbers of the local nodes I, J, K, L, M, N, O, P

The second line contains the remaining nodes:

- Field 1-12: the global numbers of the local nodes Q, R, S, T, U, V, W, X, Y, Z, A, B

The function readElementsEdges then fills then the grid data structure associated with the elements and edges.

Nodes
The nodes are defined after the NBLOCK command. The format is:

- Field 1: node number
- Field 2,3: - [3]
- Field 4-6: the coordinates
- Field 7-9: - [3]

Zero entries at the end are omitted. The coordinates for the nodes are then saved by the function readCoord;

Boundary conditions
The boundary conditions are at the end of the cdb-file after an empty line. In the case of zero boundary conditions the format is

- Field 1: "D"
- Field 2: node
- Field 3: dof label (AZ, VOLT)
- Field 4,5: dof value (0)

[3] the information in these fields is not used

Physical Parameter
The physical parameters are represented by the following entries:

- HARFREQ for $f = 2\pi\omega$
- MURX for μ_{rx}
- RSVX for $1/\sigma$

3.4.2 HB File

ANSYS saves the Galerkin matrix **A** and the rhs in the Harwell Boeing format. The HB format was created to exchange sparse matrices in text format. HB uses the CCS (Compressed Column Storage) format, which is similar to the CRS (Compressed Row Storage) format, the difference is that it is column oriented instead of row oriented. The text file starts with a header block, which contains information about the storage format and the required memory to save the data. The header block is followed by several data blocks containing the matrix and the rhs. The first data block contains the column start pointers, followed by blocks with the row indices and the numerical values of the matrix entries. The last data block contains the rhs.

To read the data from the file I use the open source code [11] . The code I use is slightly modified because the original version used single precision and had no support for complex matrices. The interface to the code is realized in the function readMatrixRHS(). There are several outputs corresponding to the entries in HB-file. The most important are:

- mxtype: a three character array describing the storage format. In our case it should be 'CSA' (Complex, Symmetric, Assembled)
- nnzero: number of nonzeros in the matrix. Note that in case of a symmetric matrix HB only saves (and counts) the lower left triangle.
- ncol, nrow: the dimension of the matrix
- colptr: int array of length ncol +1, colptr[i−1]: pointer to the first entry in column i
- rowind: int array of length nnz, rowind[i−1]: row index of entry j
- val_real, val_imag: double array of length nnz, val[i−1] : numerical value of entry j
- rhs_real, rhs_imag: double arrays of length nrow representing the rhs

To use the matrix and the rhs we have to save them in a format, that is understandable by Diffpack. In the case of the rhs the task is trivial. For the matrix it is more complicated: The first problem is that Diffpack uses the CRS-format instead of the column oriented format. Secondly Diffpack does not

support symmetric sparse matrices (at least not yet), so we have to copy all entries (with the exception of the diagonal entries). The function that builds a Diffpack sparse matrix based on the colptr, rowind and val arrays is called hb_2_dpsparse.

3.4.3 mapping File

This file lists for every equation the associated node/edge. The format is:

- Field 1: equation
- Field 2: node/edge
- Field 3: dof label (AZ, VOLT)

The routine readDOFs creates the DegFree object (cf. A.5) for the edge space.

3.5 Preconditioner

The AMS preconditioner is implemented in the class PrecAMS. The class has to implement the function applyPrec(**const** LinEqvector& r, LinEqvector& c)., which gets called in each iteration of the Krylov solver (cf. A.3). To perform the algorithm outlined in chapter 2.4.2 the preconditioner needs the following information:

- the matrix of the linear system $\mathbf{A} = \mathbf{A}_R + \mathbf{A}_I$, or in the complex case

$$\mathbb{A} = \begin{pmatrix} \mathbf{A}_R & -\mathbf{A}_I \\ -\mathbf{A}_I & -\mathbf{A}_R \end{pmatrix} \tag{3.6}$$

  in either case I will actually only use the matrix $\mathbf{A}_B = \mathbf{A}_R + \mathbf{A}_I$.

- the grid describing the domain, including all the edges (cf. A.4)
- α and β: vectors, where the entry i corresponds to the value of α, β for material i
- edge_dof: the mapping equation $\leftrightarrow$ edge

These objects are passed to the preconditioner through the constructor. Before the preconditioner is ready to use (i.e. the apply function does what it is supposed to do) we have to call the init function of the preconditioner.

3.5.1 Init

The init function builds the following matrices

- the transfer matrices $\mathbf{G}$ and $\mathbf{P} = \begin{pmatrix} \mathbf{P}_x & \mathbf{P}_y & \mathbf{P}_z \end{pmatrix}$

- the Poisson matrices Δ, $\mathbf{\Delta} = \begin{pmatrix} \widetilde{\Delta} & & \\ & \widetilde{\Delta} & \\ & & \widetilde{\Delta} \end{pmatrix}$ or alternatively the matrices $\mathbf{G}^T\mathbf{A}_B\mathbf{G}$, $\mathbf{P}_i^T\mathbf{A}_B\mathbf{P}_i$ (i=1,2,3) in the nodal space.

To create these matrices I will use three objects of the type DegFree (cf. A.5):

- edge_dof: is associated with the edge space and is created from the *.mapping file
- node_ipol_dof: is associated with the scalar Poisson problem $-\mathrm{div}\,(\alpha\ \mathbf{grad}\cdot)$ discretized on $S_{0,h}$ in the nodal space. Assigns a dof to all nodes except the boundary nodes (assuming zero boundary conditions)
- node_pot_dof: is associated with the scalar Poisson problem $-\mathrm{div}\,(\beta\ \mathbf{grad}\cdot)$ discretized on $S_{0,h}$ in the nodal space. Assigns a dof to all nodes, that are not boundary nodes and are not exclusively in a material with $\beta = 0$.

Note that if $\beta > 0$ everywhere node_ipol_dof and node_pot_dof are identical.

Building the Transfer Matrices G and P
Building the discrete gradient matrix $\mathbf{G}$ and the interpolation matrix $\mathbf{P} = \begin{pmatrix}\mathbf{P}_x\ \mathbf{P}_y\ \mathbf{P}_z\end{pmatrix}$ is quite simple with the created data structures. The matrix $\mathbf{G}$ is a mapping from the nodal space described by node_pot_dof to the edge space described by edge_dof. Similarly $\mathbf{P}_i$ is a mapping from the nodal space described by node_ipol_dof to the edge space. It is sufficient to describe the transfers between nodes and edges that represent dofs in their respective spaces. This leads to zero rows in the matrix $\mathbf{G}$. To implement those correctly one has to remember that the CRS format requires that row(i) for a empty row i points to the next nonzero entry in the matrix.

```
void PrecAMS::buildGMatrix(){

  int i;                  // equation number in edge space
  int e;              // edge associated with equation i
  int sn, en;             // sn, en: nodes s.t.: e = (sn, en)

  const int nrow = edge_dof.getNoEqu();          // number of
     rows of G (equal to number of edge equations);
  const int ncol = node_pot_dof.getNoEqu();     // number of
     columns of G (equal to number of node equations);
  int nnz = 0;                                    // number of
     nonzero entries in G
  for (i=1; i<=nrow; i++){
      e = edge_dof.getEdge(i);
      sn = grid.getNode1(e);
      en = grid.getNode2(e);
      nnz += node_pot_dof.getEqu(sn)!=0;
      nnz += node_pot_dof.getEqu(en)!=0;
```

```
    }

    Handle(SparseDS) pattern;
    pattern.rebind(new SparseDS(nrow, ncol, nnz));
    G.redim(*pattern);
    G.fill(0);          // should be redundant

    pattern->irow(1) = 1 ;

    int ij = 0;
    for (i=1; i<=nrow; i++){

        e = edge_dof.getEdge(i);

        sn = grid.getNode1(e);
        en = grid.getNode2(e);

        if (node_pot_dof.getEqu(sn)){
            ++ij;
            G(ij) = -1;
            pattern->jcol(ij) = node_pot_dof.getEqu(sn);
        }
        if (node_pot_dof.getEqu(en)){
            ++ij;
            G(ij) = 1;
            pattern->jcol(ij) = node_pot_dof.getEqu(en);
        }
        pattern->irow(i+1) = ij+1;
    }
}

void PrecAMS::buildPMatrix(){

    int i;              // equation number in edge space
    int e;              // edge associated with equation i
    int sn, en;         // sn, en: nodes s.t.: e = (sn, en)

    double tr_weight_x, tr_weight_y, tr_weight_z; // transfer
        weight

    const int nrow = edge_dof.getNoEqu();       // number of
        rows of P_x/y/z (equal to number of edge equations);
    const int ncol = node_ipol_dof.getNoEqu(); // number of
        columns of P_i (equal to number of node equations);

    int nnz = 0;        // number of nonzero entries in P
    for (int i=1; i<=nrow; i++){
        e = edge_dof.getEdge(i);
        sn = grid.getNode1(e);
        en = grid.getNode2(e);
        nnz += node_ipol_dof.getEqu(sn)!=0;
```

```
        nnz += node_ipol_dof.getEqu(en)!=0;
    }

    Handle(SparseDS) pattern;
    pattern.rebind(new SparseDS(nrow, ncol, nnz));

    Px.redim(*pattern);
    Py.redim(*pattern);
    Pz.redim(*pattern);

    Px.fill(0);
    Py.fill(0);
    Pz.fill(0);

    pattern->irow(1) = 1 ;

    int ij=0;
    for (i=1; i<=nrow; i++){

        e = edge_dof.getEdge(i);

        sn = grid.getNode1(e);
        en = grid.getNode2(e);

        tr_weight_x = .5*(grid.getCoor(en,1)-grid.getCoor(sn
            ,1));
        tr_weight_y = .5*(grid.getCoor(en,2)-grid.getCoor(sn
            ,2));
        tr_weight_z = .5*(grid.getCoor(en,3)-grid.getCoor(sn
            ,3));

        if (node_ipol_dof.getEqu(sn)){
            ++ij;
            Px(ij) = tr_weight_x;
            Py(ij) = tr_weight_y;
            Pz(ij) = tr_weight_z;
            pattern->jcol(ij) = node_ipol_dof.getEqu(sn);;
        }
        if (node_ipol_dof.getEqu(en)){
            ++ij;
            Px(ij) = tr_weight_x;
            Py(ij) = tr_weight_y;
            Pz(ij) = tr_weight_z;
            pattern->jcol(ij) = node_ipol_dof.getEqu(en);;
        }

        pattern->irow(i+1) = ij+1;
    }
}
```

FEM-matrices Δ, $\boldsymbol{\Delta}$
The matrices associated with the Poisson problems $-\text{div}\,(\beta\ \mathbf{grad}\cdot)$ and $-\text{div}\,(\alpha\ \mathbf{grad}\cdot)$ discretized on $S_{0,h}$, $(S_{0,h})^3$ are created with a subclass of the Diffpack class FEM. The FEM class has already implemented several algorithms that are typically used in finite element programs. In the default version I only have to implement the integrands function. This functions adds the contribution of an integration point to the element matrix. The value of alpha and beta for the integration point can be obtained through the material number of the element (function: grid.getMaterialType(**int** elem). The matrix assembly is then done by the function makeSystem. I always compute the full matrix (i.e. one equation for every node) and then extract the relevant submatrix based on the information in the corresponding DegFree object.

Matrices $\mathbf{P}^T\mathbf{A}\mathbf{P}$, $\mathbf{G}^T\mathbf{A}\mathbf{G}$
The block matrix $\boldsymbol{\Delta}$ can be replaced by $\mathbf{P}^T\mathbf{A}_B\mathbf{P}$ or by

$$\begin{pmatrix} \mathbf{P}_x^T\mathbf{A}_B\mathbf{P}_x & & \\ & \mathbf{P}_y^T\mathbf{A}_B\mathbf{P}_y & \\ & & \mathbf{P}_z^T\mathbf{A}_B\mathbf{P}_z \end{pmatrix} \tag{3.7}$$

The multiplication of two sparse matrices is usually not implemented in numerical libraries like Diffpack, since in general the product of two sparse matrices can be completely dense. In our case however a realization in $\mathcal{O}(nnz)$ is possible. For the entry (l,m) of $\left(\mathbf{P}_x^T\mathbf{A}_B\mathbf{P}_x\right)$ we get the following expression:

$$\left(\mathbf{P}_x^T\mathbf{A}_B\mathbf{P}_x\right)_{(l,m)} = \sum_{i=1}^{N}\sum_{j=1}^{N}\mathbf{P}_{x,(j,l)}\mathbf{A}_{B,(i,j)}\mathbf{P}_{x,(i,m)} \tag{3.8}$$

The term $\mathbf{P}_{x,(j,l)}\mathbf{A}_{B,(i,j)}\mathbf{P}_{x(i,m)}$ is equal to zero, unless the edges i, j and the nodes l, m are all part of the same element. This allows to compute the matrix $\left(\mathbf{P}_x^T\mathbf{A}_B\mathbf{P}_x\right)$ by adding up the contributions from the different elements. This is implemented in the routine build_PAP_GAG (cf. A.6). First I create the sparsity pattern of the resulting matrix, it is identical to a sparsity pattern of a FEM-matrix discretized in the nodal space with standard locally supported basis functions and can be created with the Diffpack function makeSparsityPattern from FEM.h. Similarly to a finite element assembly I will also work with local element matrices, this is done to minimize costly matrix accesses of the form (i,j).

The matrix Δ can also be replaced by $\mathbf{G}^T\mathbf{A}_B\mathbf{G}$. With exact arithmetic the two matrices would be identical ($\mathbf{G}^T\mathbf{A}_R\mathbf{G} = \mathbf{0}$, $\mathbf{G}^T\mathbf{A}_I\mathbf{G} = \Delta$. But numerical experiments show that the matrices can be quite different and that $\mathbf{G}^T\mathbf{A}_B\mathbf{G}$ yields better results for the AMS preconditioner.

3.5.2 Apply

The Krylov solver calls in each iteration the apply() function of the preconditioner. This function realizes the preconditioner action $\vec{c} = \mathbf{B}\vec{r}$ or $\begin{pmatrix} \vec{c_1} \\ \vec{c_2} \end{pmatrix} = \begin{pmatrix} \mathbf{B} & \\ & \mathbf{B} \end{pmatrix} \begin{pmatrix} \vec{r_1} \\ \vec{r_2} \end{pmatrix}$ in the complex case, where $\mathbf{B}$ is the AMS preconditioner described in chapter 2.4.2.

```
void AuxSpacePrec::apply (const LinEqVector &r, LinEqVector &c
    , TransposeMode tpmode){

    // realizes preconditioner c = Br

    no_calls++;
    cout << "apply Preconditioner, number of calls: " <<
        no_calls <<  endl;


    if (!complex){

        const Vec(dpreal)& rvec = r.getVec(1);
        Vec(dpreal)& cvec = c.getVec(1);

        applyMultPrec(*A_B, rvec, cvec);
        //applyAddPrec(*A_B, rvec, cvec);

    }
    else{

        const Vec(dpreal)& rvec1 = r.getVec(1);
        const Vec(dpreal)& rvec2 = r.getVec(2);
        Vec(dpreal)& cvec1 = c.getVec(1);
        Vec(dpreal)& cvec2 = c.getVec(2);

        applyMultPrec(*A_B, rvec1, cvec1);
        applyMultPrec(*A_B, rvec2, cvec2);
        //applyAddPrec(*A_B, rvec1, cvec1);
        //applyAddPrec(*A_B, rvec2, cvec2);

    }
}
```

Additive Variant

```
void PrecAMS::applyAddPrec(MatSparse(dpreal)& A_sys, const Vec
    (dpreal) &r, Vec(dpreal) &c){

    c2.fill(0);
    A_sys.SSOR1it(c, c2, r, 1);
```

```
    if (!two_level_method){ // beta != 0
        G.prod(r, rhs_beta, TRANSPOSED);          // rhs = G^T
            * r
        sys_beta->attach(x_beta, rhs_beta);
        solver->solve(*sys_beta);                 // x = D^-1 *
             rhs
        G.prod(x_beta, c, NOT_TRANSPOSED, true); // c = c + G*
            x
    }

    Px.prod(r, rhs, TRANSPOSED);                  // rhs = Px^T
        * r
    sys_x->attach(x, rhs);
    solver->solve (*sys_x);                       // x = Dx^-1 *
         rhs
    Px.prod(x, c, NOT_TRANSPOSED, true);          // c = c + Px
        * x

    Py.prod(r, rhs, TRANSPOSED);
    sys_y->attach(x, rhs);
    solver->solve (*sys_y);
    Py.prod(x, c, NOT_TRANSPOSED, true);

    Pz.prod(r, rhs, TRANSPOSED);
    sys_z->attach(x, rhs);
    solver->solve (*sys_z);
    Pz.prod(x, c, NOT_TRANSPOSED, true);


}
```

Multiplicative Variant

```
void PrecAMS::applyMultPrec(MatSparse(dpreal)& A_sys, const
    Vec(dpreal) &r, Vec(dpreal) &c){

    c.fill(0.0);                    //c=0;

    A_sys.SSOR1it(c2, c, r, 1); // apply a symmetric Gauss
        Seidel sweep to Ac = r;

    if (!two_level_method){ // beta!=0
        computeRes(res, c2, r, A_sys);        // res = r-Ac

        G.prod(res, rhs_beta, TRANSPOSED);  // rhs = G^T*res
        sys_beta->attach(x_beta, rhs_beta); // x = D^-1 * rhs
        solver->solve(*sys_beta);
        G.prod(x_beta, c2, NOT_TRANSPOSED, true); // c = c + G
            *x
```

```
    }

    computeRes(res, c2, r, A_sys);

    Px.prod(res, rhs, TRANSPOSED);              // rhs = Px^T
    sys_x->attach(x, rhs);
    solver->solve (*sys_x);                     // x = Dx^-1 * rhs
    Px.prod(x, c2, NOT_TRANSPOSED, true);       // c = c + (P*x)_x

    Py.prod(res, rhs, TRANSPOSED);
    sys_y->attach(x, rhs);
    solver->solve (*sys_y);
    Py.prod(x, c2, NOT_TRANSPOSED, true);

    Pz.prod(res, rhs, TRANSPOSED);
    sys_prec_z->attach(x, rhs);
    solver->solve (*sys_z);
    Pz.prod(x, c2, NOT_TRANSPOSED, true);

    if (!two_level_method){
        computeRes(res, c2, r, A_sys);

        G.prod(res, rhs_beta, TRANSPOSED);
        sys_beta->attach(x_beta, rhs_beta);
        solver->solve(*sys_beta);
        G.prod(x_beta, c2, NOT_TRANSPOSED, true);
    }

    A_sys.SSOR1it(c, c2, r, 1);


}
```

Remarks:

- For every call of the preconditioner we have to solve four (or in the complex case eight) problems in the nodal space. To achieve overall optimal performance an approximative solution with AMG V-cycles would be optimal. Unfortunately caused by unexpected problems with the AMG solver I was not able to implement that in time. Instead I use a direct solver.
- A symmetric Gauss-Seidel sweep is a special case of SSOR iteration with ω set to one. The SSOR iteration is implemented in Diffpack by the function SSOR1it. So all we have to do to realize the smoothing step is to call this function with $\omega = 1$.

4 Testcases, Results

We investigate the performance of the following variants of the AMS preconditioner:

1. Additive variant with Poisson auxiliary matrices Δ, $\mathbf{\Delta}$
2. Multiplicative variant with Poisson auxiliary matrices Δ, $\mathbf{\Delta}$
3. Additive variant with auxiliary matrices $\mathbf{P}_i^T\mathbf{A}\mathbf{P}_i$ $(i = 1, 2, 3)$, $\mathbf{G}^T\mathbf{A}\mathbf{G}$
4. Multiplicative variant with auxiliary matrices $\mathbf{P}_i^T\mathbf{A}\mathbf{P}_i$ $(i = 1, 2, 3)$, $\mathbf{G}^T\mathbf{A}\mathbf{G}$

4.1 Constant Coefficients

As a first example we consider the unit cube meshed with tetrahedral elements.

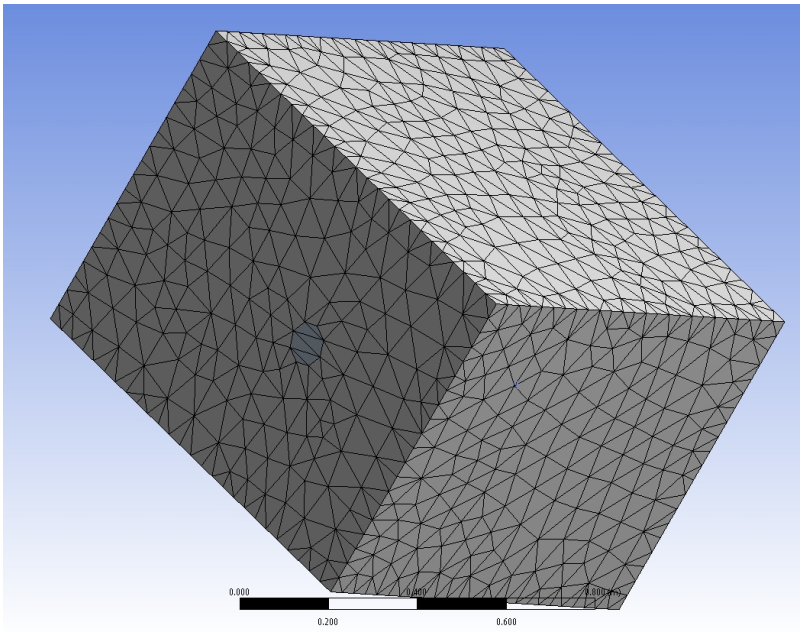


Figure 3: unit cube

The material parameters are constant in the domain and have the following values:
- $\sigma = 10^6$
- $\mu = \mu_o\mu_{rx}$, with $\mu_0 = 4\pi 10^{-7}$, $\mu_{rx} = 1$
The frequency is 1. With these parameters we get for α and β:

- $\alpha = \frac{1}{4\pi} \cdot 10^7$
- $\beta = 2\pi 10^6$

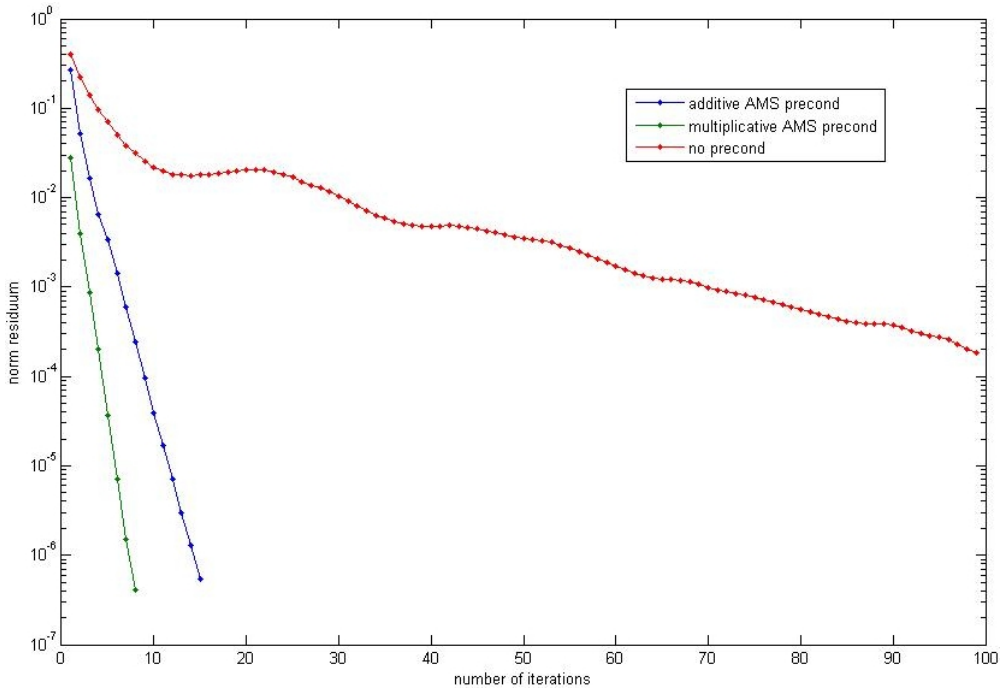


Figure 4: relative error of the residuum against the number of iterations

The following tables show the performance of the preconditioner with the convergence tolerance set to 10^{-6}

N	n_{it} 1	n_{it} 2	n_{it} 3	n_{it} 4
7673	15	8	16	8
17794	14	8	16	8
34428	15	8	16	8
73372	15	8	16	8
142072	15	8	17	8

Table 1: number of iteration for the real case

N	n_{it} 2	n_{it} 4
7673	31	28
17794	33	29
34428	35	29
73372	37	30
142072	39	30

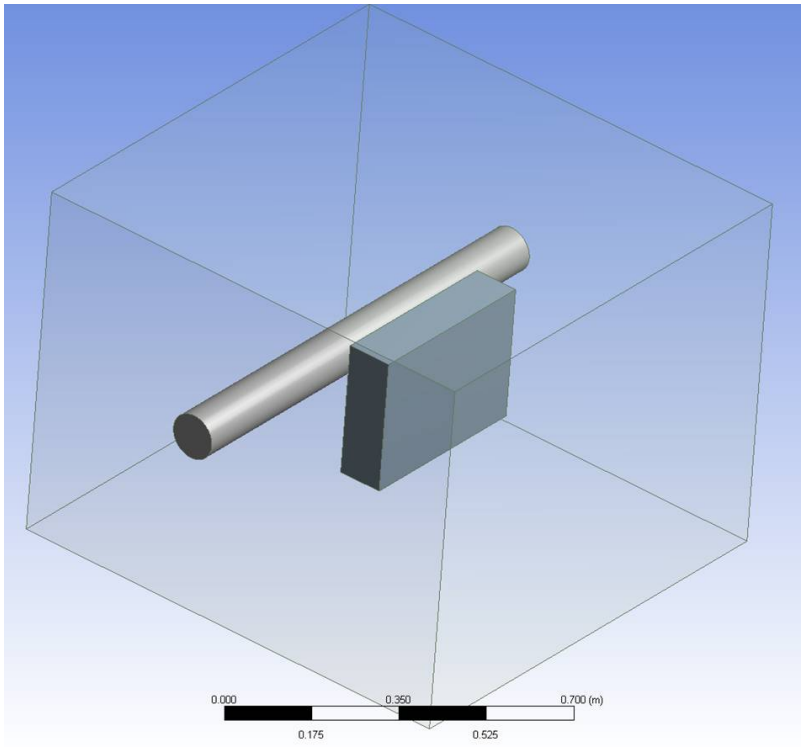
Table 2: number of iteration for the complex case

N	n_{it}	init	transfer	residuum	smoother	solve
7673	31	0.087	0.02	0.03	0.016	
17794	33	0.15	0.028	0.032	0.048	
34428	35	0.30	0.065	0.015	0.281	
73372	37	0.55	0.10	0.062	0.405	
142072	39	0.91	0.18	0.181	0.813	

Table 3: time spent in different parts of the preconditioner (variant 4) on the coarsest mesh (Core 2 Duo 2.8GHz)

4.2 Singular Problem

The geometry of the second example is shown in the following picture



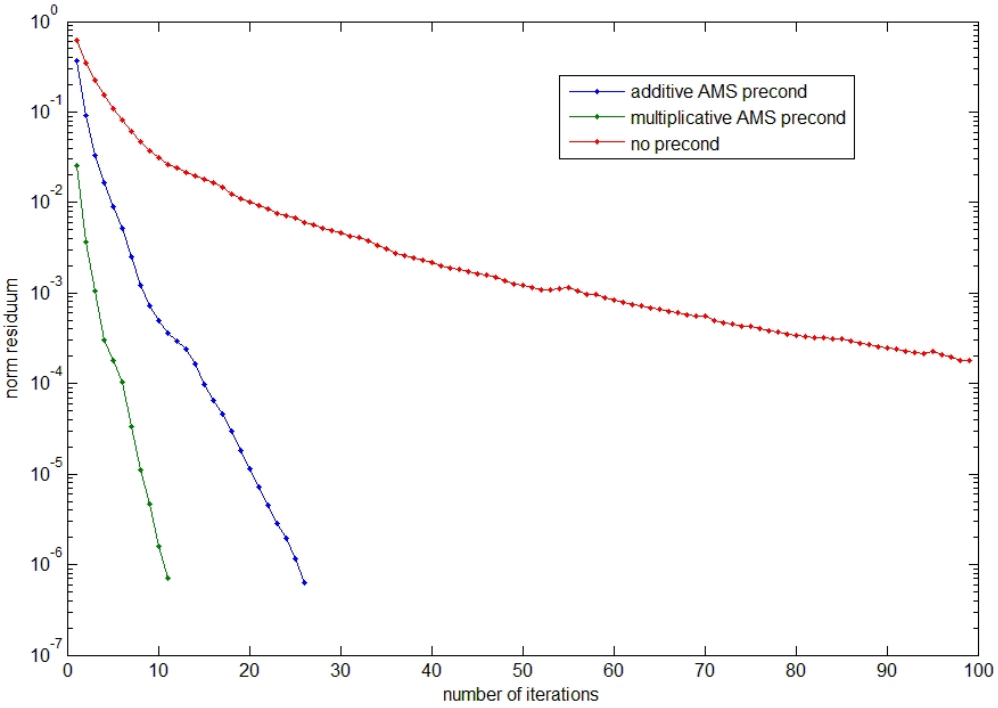
The model consists of three different materials with the following parameters:

- air:
  - $\sigma = 0$
  - $\mu = \mu_o \mu_{rx}$, with $\mu_0 = 4\pi 10^{-7}$, $\mu_{rx} = 1$
- wall:
  - $\sigma = 10^7$

- $\mu = \mu_o \mu_{rx}$, with $\mu_0 = 4\pi 10^{-7}$, $\mu_{rx} = 1$

- conductor:
  - $\sigma = 10^6$
  - $\mu = \mu_o \mu_{rx}$, with $\mu_0 = 4\pi 10^{-7}$, $\mu_{rx} = 200$

Note that β_{air} is zero. The matrix of the system is semi-definit and the rhs has to satisfy compatibility conditions.



N	n_{it} 4	n_{it} 3
5508	9	21
6570	10	25
7418	11	26
10665	12	27
21027	12	25
49505	12	26

Table 4: number of iteration for the real case

The complex case did not converge for that example.

5 Conclusions

The numerical tests for the real case demonstrate that the AMS preconditioner works very well even for challenging problems. Theoretical results indicate that it should also work for complex-valued systems. The numerical tests however show mixed results . Further tests are needed to explain these results and hopefully improve the performance for complex-valued systems.

A comparison between different variants of the preconditioner indicated that the multiplicative variant with matrices $\mathbf{P}^T\mathbf{AP}$, $\mathbf{G}^T\mathbf{AG}$ for the nodal space problems is the best choice.

The following extensions of the application could be implemented in the future:

- support for more general boundary conditions
- support for different elements (hexahedron, pyramid, prism)
- replace the direct solver in the nodal space by AMG V-cycles

A Appendix

A.1 Preconditioner for Complex Systems

Consider a complex linear system of equations:

$$\mathbf{A}z = b \tag{A.1}$$

with

$$\begin{aligned} \mathbf{A} &= \mathbf{A}_R + i\mathbf{A}_I \quad \mathbf{A}_R, \mathbf{A}_I \in \mathbb{R}^{n,n} \\ b &= b_R + ib_I \quad b_R, b_I \in \mathbb{R}^n \\ \vec{z} &= \vec{z}_R + iz_I \quad z_R, \vec{z}_I \in \mathbb{R}^n \end{aligned}$$

Assumption: $\mathbf{M} \in \mathbb{R}^{n,n}$ s.p.d. with

$$\gamma z^H \mathbf{M} z \leq \left| z^H \mathbf{A} z \right| \leq \tau z^H \mathbf{M} z \quad \forall z \in \mathbb{C}^n \tag{A.2}$$

then we get for the preconditioned system

$$\begin{pmatrix} \mathbf{M}^{-1/2} & \\ & -\mathbf{M}^{-1/2} \end{pmatrix} \begin{pmatrix} \mathbf{A}_R & -\mathbf{A}_I \\ -\mathbf{A}_I & -\mathbf{A}_R \end{pmatrix} \begin{pmatrix} \mathbf{M}^{-1/2} & \\ & -\mathbf{M}^{-1/2} \end{pmatrix} = \widetilde{\mathbb{A}} \tag{A.3}$$

the spectral condition number:

$$\kappa = \left(\frac{\tau}{\gamma} \right) \tag{A.4}$$

Proof:
If $\mathbf{A}z = b$ then

$$\gamma z^H \mathbf{M} z \leq \left| z^H \mathbf{A} z \right| = \left| z^H b \right| = \left| z^H \mathbf{M}\mathbf{M}^{-1} b \right| \leq \left(z^H \mathbf{M} z \right)^{1/2} \left(b^H \mathbf{M}^{-1} b \right)^{1/2}$$

$$\Rightarrow \|z\|_M \leq \frac{1}{\gamma} \|b\|_{M^{-1}} \tag{A.5}$$

$$\gamma w^H w \leq \left| w^H \underbrace{\mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2}}_{\widetilde{\mathbf{A}}} w \right| \leq \tau w^H w \quad \forall w \in \mathbb{C}^n$$

If $\widetilde{\mathbf{A}} w = \widetilde{b}$ then

$$\|w\| \leq \frac{1}{\gamma} \left\| \widetilde{b} \right\| \tag{A.6}$$

$$\widetilde{\mathbf{A}} w = \widetilde{b} \Leftrightarrow \begin{pmatrix} \widetilde{\mathbf{A}}_R & -\widetilde{\mathbf{A}}_I \\ -\widetilde{\mathbf{A}}_I & -\widetilde{\mathbf{A}}_R \end{pmatrix} \begin{pmatrix} w_R \\ w_I \end{pmatrix} = \begin{pmatrix} \widetilde{b}_R \\ -\widetilde{b}_I \end{pmatrix} \tag{A.7}$$

$$\Rightarrow \left\| \begin{pmatrix} w_R \\ w_I \end{pmatrix} \right\| \leq \frac{1}{\gamma} \left\| \begin{pmatrix} \widetilde{b}_R \\ -\widetilde{b}_I \end{pmatrix} \right\| \Rightarrow \left\| \widetilde{\mathbb{A}}^{-1} \right\| \leq \frac{1}{\gamma} \tag{A.8}$$

Assumption:

$$\left| z^H \mathbf{A} w \right| \leq \tau \|z\|_M \|w\|_M \tag{A.9}$$

$$\Rightarrow \left|z^H \widetilde{\mathbf{A}} w\right| \leq \tau \|z\| \|w\| \Rightarrow \left\|\widetilde{\mathbf{A}}\right\| \leq \tau$$

If $\widetilde{\mathbf{A}}\vec{w} = \widetilde{\vec{b}}$ then

$$\left\| \begin{pmatrix} \widetilde{\vec{b}}_R \\ -\widetilde{\vec{b}}_I \end{pmatrix} \right\| = \left\| \widetilde{\mathbb{A}} \begin{pmatrix} w_R \\ w_I \end{pmatrix} \right\| \leq \tau \tag{A.10}$$

$$\left\|\widetilde{\mathbb{A}}\right\| \leq \tau \tag{A.11}$$

Special case $\mathbf{A}_R, \mathbf{A}_I$ symmetric, positive $\mathbf{A}_I$ s.p.d.

$$\begin{aligned}
\left|z^H \mathbf{A} z\right| &= \left|z^H \mathbf{A}_R z + i z^H \mathbf{A}_I z\right| && \text{(A.12)} \\
&\geq \frac{1}{\sqrt{2}} \left(\left|z^H \mathbf{A} z\right| + \left|z^H \mathbf{A} z\right|\right) && \text{(A.13)} \\
&\geq \frac{1}{\sqrt{2}} \left(z^H \mathbf{A}_R z + z^H \mathbf{A}_I z\right) && \text{(A.14)} \\
&\geq \frac{1}{\sqrt{2}} z^H \underbrace{\left(\mathbf{A}_R + \mathbf{A}_I\right)}_{\mathbf{M}} z && \text{(A.15)}
\end{aligned}$$

$$\begin{aligned}
\left|z^H \mathbf{A} w\right| &= \left|z^H \mathbf{A}_R w + i z^H \mathbf{A}_I w\right| && \text{(A.16)} \\
&\leq \left(\left|z^H \mathbf{A} w\right| + \left|z^H \mathbf{A} w\right|\right) && \text{(A.17)} \\
&\leq \sqrt{2}\sqrt{\left|z^H \mathbf{A} w\right|^2 + \left|z^H \mathbf{A} w\right|^2} && \text{(A.18)} \\
&\leq \sqrt{2}\sqrt{\left(z^H \mathbf{A}_R z\right)\left(w^H \mathbf{A}_R w\right) + \left(z^H \mathbf{A}_I z\right)\left(w^H \mathbf{A}_I w\right)} && \text{(A.19)} \\
&\leq \left(z^H \mathbf{M} z\right)^{1/2} \left(w^H \mathbf{M} w\right)^{1/2} && \text{(A.20)}
\end{aligned}$$

With $\mathbf{M} = \mathbf{A}_R + \mathbf{A}_I$ we have $\gamma = \frac{1}{2}\sqrt{2},\ \tau = 1$

A.2 Sparse Matrices in Diffpack

The storage of sparse matrices is implemented in the Diffpack class MatSparse. The class uses the Compressed Row Storage (CRS) format. For a complete documentation see [9].

A.3 Linear Systems in Diffpack

Linear systems are represented in Diffpack by the class LinEqSystemStd [9]. The class contains a blockmatrix of the type LinEqMatrix [9] and two blockvectors of the type LinEqVector [9] representing the rhs and the solution of the system.

To solve a preconditioned linear systems we can use the class LinEqSystemPrec [9]. This class is a simple extension of the class LinEqSystemStd and has additionally a preconditioner attached. The preconditioner has to implement the function applyPrec(**const** LinEqVector& r, LinEqVector& c). To solve the system we then have to create a solver object and call its solve function with the linear system as argument.

A.4 Grid in Diffpack

The necessary data structures to describe a grid are realized in the Diffpack class GridFE (cf.). The class contains a matrix nodel, which lists the nodes for every element. The coordinates for a node are saved in the matrix coor. Furthermore the class marks every boundary node and has a vector material_type which contains for every element its material number (1,2, .., nmat). The physical data corresponding to a material has to be saved outside of the GridFE object. Accessing this information can be done through the following functions (for a complete list of all the functions see the man page of the class [9])

- loc2glob(**int** e, **int** local_node)
  given a local node number in an element, the function returns the corresponding global number of that node.
- getCoor(**int** node, **int** dir)
  get the coordinates of a node.
- getMaterialType (**int** e)
  gets the material number of an element.
- boNode (**int** node)
  returns "true" if a given node is marked with a boundary indicator.

To implement the AMS preconditioner we need two additional matrices describing the (oriented) edges of the mesh. The first one - edgel - is similar to nodel and lists the edges for every element. An oriented edge is described by its head and tail node, this information is saved in the matrix edges.

- loc2globEdge(**int** e, **int** local_edge)
  given a local edge number in an element, the function returns the corresponding global number of that edge.

- getNode1(**int** edge)
  get the tail node of an edge.

- getNode2(**int** edge)
  get the head node of an edge.

A.5 DegFree

This class realizes the mappings: node/edge $\mapsto$ equation and equation $\mapsto$ node/edge. This information is saved in two arrays and can be accessed as follows:

- **int** getEdge(**int** dof)
  get the edge associated with dof

- **int** getNode(**int** dof)
  get the node associated with dof (just a different name for the function getEdge).

- **int** getDof(**int** node/edge)
  get the dof associated with edge/node. returns 0 if there is no dof associated with the edge/node

A.6 Matrices $\mathbf{P}^T\mathbf{AP}$, $\mathbf{G}^T\mathbf{AG}$

```
void PrecAMS::build_PAP_GAG(){

    // create sparsity pattern for nodal space
    Handle(SparseDS)          pattern;
    Handle(DegFreeFE)         dof;
    pattern.rebind(new SparseDS());
    dof.rebind (new DegFreeFE(grid, 1));
    makeSparsityPattern(*pattern, *dof );

    const int n_node_el = 4;     // number of nodes per element
    const int n_edge_el = 6;     // number of edges per element
    const int nelem = grid.getNoElms();

    int e1, e2;           // local edgenumber
    int gl_e1, gl_e2;     // global edgenumber
    int dof_e1, dof_e2;       // equation associated with edge
        gl_e1, gl_e2
    int sn_e1, en_e1;     // first and second node for edge e1:
        gl_e1 = (sn_e1, en_e1)
    int sn_e2, en_e2;     // first and second node for edge e2:
        gl_e2 = (sn_e2, en_e2)
```

```
int loc_node_1, loc_node_2;
int sn_e1_loc, en_e1_loc;    // first and second local node
    for edge e1
int sn_e2_loc, en_e2_loc;    // first and second local node
    for edge e2

double  tr_weight_x, tr_weight_y, tr_weight_z;

Mat(dpreal) Ae_x, Ae_y, Ae_z;
Ae_x.redim(n_node_el,n_node_el); // local elem mat
Ae_y.redim(n_node_el,n_node_el); // local elem mat
Ae_z.redim(n_node_el,n_node_el); // local elem mat
Mat(dpreal) Ae;
Ae.redim(n_node_el,n_node_el);

MatSimplest(int) loc_node;
loc_node.redim(6,2);
loc_node(1,1) = 1; loc_node(1,2) = 2;
loc_node(2,1) = 2; loc_node(2,2) = 3;
loc_node(3,1) = 1; loc_node(3,2) = 3;
loc_node(4,1) = 1; loc_node(4,2) = 4;
loc_node(5,1) = 2; loc_node(5,2) = 4;
loc_node(6,1) = 3; loc_node(6,2) = 4;

VecSimple(int) idx(n_node_el);       // local node to
    global node mapping
int index;

Lx.redim(*pattern);       // Px^TAPx
Ly.redim(*pattern);       // Py^TAPy
Lz.redim(*pattern);       // Pz^TAPz
Lx.fill(0);
Ly.fill(0);
Lz.fill(0);
L.redim(*pattern);        // G^TAG
L.fill(0);
VecSimple(bool) alr_transferred(A_B->getNoNonzeroes());
alr_transferred.fill(false);

dpreal tr_val;
dpreal tr_val_x, tr_val_y, tr_val_z;

for (int i=1; i<=nelem; i++){

    Ae_x.fill(0);
    Ae_y.fill(0);
    Ae_z.fill(0);
    Ae.fill(0);

    for (int j=1; j<=n_node_el; j++){
```

```
    idx(j) = grid.loc2glob(i,j);
}

for (e1=1; e1<=n_edge_el; e1++){

    gl_e1 = grid.getEdge(i, e1);

    if ( edge_dof.getEqu(gl_e1)==0 )
        continue;

    dof_e1 = edge_dof.getEqu(gl_e1);

    sn_e1 = grid.getNode1(gl_e1);
    en_e1 = grid.getNode2(gl_e1);

    loc_node_1 = loc_node(e1,1);
    loc_node_2 = loc_node(e1,2);

    if (grid.loc2glob(i, loc_node_1) == sn_e1){
        sn_e1_loc = loc_node_1;
        en_e1_loc = loc_node_2;
    }
    else{
        sn_e1_loc = loc_node_2;
        en_e1_loc = loc_node_1;
    }

    tr_weight_x = .5*(grid.getCoor(en_e1,1)-grid.
        getCoor(sn_e1,1));
    tr_weight_y = .5*(grid.getCoor(en_e1,2)-grid.
        getCoor(sn_e1,2));
    tr_weight_z = .5*(grid.getCoor(en_e1,3)-grid.
        getCoor(sn_e1,3));

    index = A_B->idx(dof_e1, dof_e1);
    if (index==0){
        cout << "Warning in routine buildPoissonMatrix
            : Entry (" << dof_e1 <<", " << dof_e1 << ")
             not in sparsity pattern of A" << std::endl
            ;
        continue;
    }

    if (!alr_transferred(index)){

        tr_val = (*A_B)(index);

        tr_val_x = tr_val  * tr_weight_x * tr_weight_x
            ;
        tr_val_y = tr_val  * tr_weight_y * tr_weight_y
            ;
```

```
        tr_val_z = tr_val  * tr_weight_z * tr_weight_z
            ;

        Ae_x(sn_e1_loc, sn_e1_loc) += tr_val_x;
        Ae_x(sn_e1_loc, en_e1_loc) += tr_val_x;
        Ae_x(en_e1_loc, sn_e1_loc) += tr_val_x;
        Ae_x(en_e1_loc, en_e1_loc) += tr_val_x;

        Ae_y(sn_e1_loc, sn_e1_loc) += tr_val_y;
        Ae_y(sn_e1_loc, en_e1_loc) += tr_val_y;
        Ae_y(en_e1_loc, sn_e1_loc) += tr_val_y;
        Ae_y(en_e1_loc, en_e1_loc) += tr_val_y;

        Ae_z(sn_e1_loc, sn_e1_loc) += tr_val_z;
        Ae_z(sn_e1_loc, en_e1_loc) += tr_val_z;
        Ae_z(en_e1_loc, sn_e1_loc) += tr_val_z;
        Ae_z(en_e1_loc, en_e1_loc) += tr_val_z;

        Ae(sn_e1_loc, sn_e1_loc) += tr_val;
        Ae(sn_e1_loc, en_e1_loc) -= tr_val;
        Ae(en_e1_loc, sn_e1_loc) -= tr_val;
        Ae(en_e1_loc, en_e1_loc) += tr_val;

        alr_transferred(index) = true;
    }

    for (e2=e1+1; e2<=n_edge_el; e2++){

        gl_e2 = grid.getEdge(i, e2);

        if ( edge_dof.getEqu(gl_e2) ==0)
            continue;

        dof_e2 = edge_dof.getEqu(gl_e2);

        if (dof_e1<dof_e2)
            index = A_B->idx(dof_e1, dof_e2);
        else
            index = A_B->idx(dof_e2, dof_e1);

        if (index==0){
            cout << "Warning in routine
                buildPoissonMatrix: Entry (" << dof_e1
                <<", " << dof_e2 << ") not in sparsity
                pattern of A" << std::endl;
            continue;
        }
        if (!alr_transferred(index)){

            sn_e2 = egrid.getNode1(gl_e2);
            en_e2 = egrid.getNode2(gl_e2);
```

```
loc_node_1 = loc_node(e2,1);
loc_node_2 = loc_node(e2,2);

if (grid.loc2glob(i, loc_node_1) == sn_e2)
    {
     sn_e2_loc = loc_node_1;
     en_e2_loc = loc_node_2;
}
else{
     sn_e2_loc = loc_node_2;
     en_e2_loc = loc_node_1;
}

tr_val = (*A_B)(index);

tr_val_x = tr_val*tr_weight_x*.5*(grid.
    getCoor(en_e2,1)-grid.getCoor(sn_e2,1))
    ;
tr_val_y = tr_val*tr_weight_y*.5*(grid.
    getCoor(en_e2,2)-grid.getCoor(sn_e2,2))
    ;
tr_val_z = tr_val*tr_weight_z*.5*(grid.
    getCoor(en_e2,3)-grid.getCoor(sn_e2,3))
    ;

Ae_x(sn_e1_loc, sn_e2_loc) += tr_val_x;
Ae_x(sn_e1_loc, en_e2_loc) += tr_val_x;
Ae_x(en_e1_loc, sn_e2_loc) += tr_val_x;
Ae_x(en_e1_loc, en_e2_loc) += tr_val_x;

Ae_x(sn_e2_loc, sn_e1_loc) += tr_val_x;
Ae_x(sn_e2_loc, en_e1_loc) += tr_val_x;
Ae_x(en_e2_loc, sn_e1_loc) += tr_val_x;
Ae_x(en_e2_loc, en_e1_loc) += tr_val_x;

Ae_y(sn_e1_loc, sn_e2_loc) += tr_val_y;
Ae_y(sn_e1_loc, en_e2_loc) += tr_val_y;
Ae_y(en_e1_loc, sn_e2_loc) += tr_val_y;
Ae_y(en_e1_loc, en_e2_loc) += tr_val_y;

Ae_y(sn_e2_loc, sn_e1_loc) += tr_val_y;
Ae_y(sn_e2_loc, en_e1_loc) += tr_val_y;
Ae_y(en_e2_loc, sn_e1_loc) += tr_val_y;
Ae_y(en_e2_loc, en_e1_loc) += tr_val_y;

Ae_z(sn_e1_loc, sn_e2_loc) += tr_val_z;
Ae_z(sn_e1_loc, en_e2_loc) += tr_val_z;
Ae_z(en_e1_loc, sn_e2_loc) += tr_val_z;
Ae_z(en_e1_loc, en_e2_loc) += tr_val_z;
```

```
                        Ae_z(sn_e2_loc, sn_e1_loc) += tr_val_z;
                        Ae_z(sn_e2_loc, en_e1_loc) += tr_val_z;
                        Ae_z(en_e2_loc, sn_e1_loc) += tr_val_z;
                        Ae_z(en_e2_loc, en_e1_loc) += tr_val_z;


                        Ae(sn_e1_loc, sn_e2_loc) += tr_val;
                        Ae(sn_e1_loc, en_e2_loc) -= tr_val;
                        Ae(en_e1_loc, sn_e2_loc) -= tr_val;
                        Ae(en_e1_loc, en_e2_loc) += tr_val;

                        Ae(sn_e2_loc, sn_e1_loc) += tr_val;
                        Ae(sn_e2_loc, en_e1_loc) -= tr_val;
                        Ae(en_e2_loc, sn_e1_loc) -= tr_val;
                        Ae(en_e2_loc, en_e1_loc) += tr_val;

                        alr_transferred(index) = true;
                    }
                }
            }
            Lx.assemble(Ae_x, idx, idx, i);
            Ly.assemble(Ae_y, idx, idx, i);
            Lz.assemble(Ae_z, idx, idx, i);
            L.assemble(Ae, idx, idx, i);
        }
    }
```

B Overview Files

- main.cpp
- PrecAMS.h
- PrecAMS.cpp
- DriverAndData.h
- DriverAndData.cpp
- AnsysInput.h
- AnsysInput.cpp
- Poisson.h
- Poisson.cpp
- PhysParamter.h
- DegFree.h
- hb_io.h
- hb_io.cpp
- EdgeGridFE.h
- EdgeGridFE.cpp

References

[1] R. Hiptmair, J. Xu: Auxiliary Space Preconditioning for Edge Elements, IEEE Trans. Magnetics, 44 (2008), pp. 938-941

[2] R. Hiptmair, J. Xu: Nodal Auxiliary Space Preconditioning in H(curl) and H(div) spaces, SIAMJ. Numer. Anal., 45 (2007), pp. 2483-2509

[3] T. Kolev, P. Vassilevski: Parallel Auxiliary Space AMG for H(curl) Problems, J. Comp. Math., 27 (2009), pp. 604-623

[4] T. Kolev, P. Vassilevski, Parallel H1-based auxiliary space AMG solver for H(curl) problems, Technical Report UCRL-TR-222763, LLNL, Livermore, California, USA, 2006

[5] J. Xu, The auxiliary space method and optimal preconditioning techniques for unstructured grids, Computing, 56 (1996), pp. 215-235

[6] H. P. Langtangen: Computational Partial Differential Equations, Second Edition, Springer, ISBN: 3-540-43416-X

[7] — : SparseMatrices, p. 787, 793

[8] — : Chapter 3: Programming of Finite Element Solvers, p. 251 ff

[9] Online Diffpack Documentation: `http://www.diffpack.com/cgi-bin/search.cgi`

[10] Matrix Market, Text File Formats: `http://math.nist.gov/MatrixMarket/formats.html`

[11] HB_IO, Harwell Boeing Sparse Matrix Files Read and Write Utilities `http://people.sc.fsu.edu/~burkardt/cpp_src/hb_io/hb_io.html`

[12] ANSYS: Theory Reference: chapter 5.1.: Electromagnetic Field Fundamentals

[13] ANSYS: Programmer's Manual for Mechanical APDL: chapter 3.2.: Coded Database File Commands