Term Project/Semesterarbeit
(Computational Science & Engineering)

Gian-Marco Baschera / Nicolas Hodler
Supervisor: Prof. Dr. R. Hiptmair (SAM, D-MATH)

Numerical Simulation of Harmonic Map Heat Flow

Contents

1 Introduction

In this term project we present a simulation of the temporal evolution of vector fields of unit length governed by the harmonic map heat flow equation $\frac{\partial \mathbf{m}}{\partial t} = \mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m})$. In the first part (sections 2-4), we show the discretization of the equation with a finite element method which leads us to the sequential simulation code. We then study the convergence of the discretization method (section 5), and finally, we show how the simulation can be efficiently parallelized (section 6).

2 Governing equations

Harmonic map heat flow refers to the gradient flow of the Dirichlet functional for vector fields of unit length. On a given computational domain $\Omega \subset \mathbb{R}^2$ and for a given period of time $]0, T[$, $T > 0$, this results in the evolution equations for $\mathbf{m} = \mathbf{m}(t, \mathbf{x}) :]0, T[\times \Omega \mapsto \mathbb{R}^3$:

$$\begin{aligned} \frac{\partial \mathbf{m}}{\partial t} &= \mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m}) \quad \text{in }]0, T[\times \Omega \,, \\ \mathbf{m}(0) &= \mathbf{m}_0 \quad \text{in } \Omega \,, \\ \frac{\partial \mathbf{m}}{\partial \mathbf{n}} &= 0 \quad \text{on }]0, T[\times \partial\Omega \,. \end{aligned} \tag{1}$$

The underlying energy is

$$\mathcal{E}(\mathbf{m}) = \tfrac{1}{2} \int_\Omega |\nabla \mathbf{m}|^2 \, \mathrm{d}\mathbf{x} \,, \tag{2}$$

where $\nabla \mathbf{m}$ designates the Jacobi matrix of $\mathbf{m}$ and $|\nabla \mathbf{m}|$ gives its Frobenius norm. Thanks to the boundary condition on $\mathbf{m}$, we find

$$\begin{aligned} \frac{d\mathcal{E}(\mathbf{m})}{dt} &= \int_\Omega \nabla \mathbf{m} : \nabla \frac{\partial \mathbf{m}}{\partial t} \, \mathrm{d}\mathbf{x} = - \int_\Omega \Delta \mathbf{m} \cdot \frac{\partial \mathbf{m}}{\partial t} \, \mathrm{d}\mathbf{x} \\ &= - \int_\Omega \Delta \mathbf{m} \cdot (\mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m})) \, \mathrm{d}\mathbf{x} = - \int_\Omega |\Delta \mathbf{m} \times \mathbf{m}|^2 \, \mathrm{d}\mathbf{x} \,. \end{aligned} \tag{3}$$

This reveals that harmonic map heat flow is a dissipative process with respect to the energy from (2). Moreover,

$$\frac{d|\mathbf{m}|^2}{dt} = 2\mathbf{m} \cdot \frac{\partial \mathbf{m}}{\partial t} = 2\mathbf{m} \cdot (\mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m})) = 0 \,, \tag{4}$$

which shows that $|\mathbf{m}(t, \mathbf{x})| = |\mathbf{m}_0(\mathbf{x})|$ for all $(t, \mathbf{x}) =]0, T[\times \Omega$. If we fix $|\mathbf{m}_0| = 1$ almost erverywhere in Ω, which is usually done, then $|\mathbf{m}| = 1$ almost everywhere for all times.

In the sequel, let us assume that $|\mathbf{m}(t, \mathbf{x})| = 1$ for all $(t, \mathbf{x}) =]0, T[\times \Omega$ ("saturation").

3 Discretization

3.1 Timestepping

For the temporal discretization of (1) the method of Heun can be employed. We use an equidistant grid in time and try to compute $\mathbf{m}(t)$ for instances $t_n := nk$, $k = {}^{T}/_{M}$. We introduce the notations

$$\mathbf{m}^n \approx \mathbf{m}(t_n) \quad , \quad \delta_t \mathbf{m}^{n+1/2} \approx \frac{\mathbf{m}^{n+1} - \mathbf{m}^n}{k} \quad , \quad \overline{\mathbf{m}}^{n+1/2} \approx \tfrac{1}{2}(\mathbf{m}^{n+1} + \mathbf{m}^n) \; .$$

Then the discrete evolution can be stated as

$$\delta_t \mathbf{m}^{n+1/2} = \overline{\mathbf{m}}^{n+1/2} \times (\Delta \overline{\mathbf{m}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}) \; , \quad \mathbf{m}^0 = \mathbf{m}_0 \quad \text{in } \Omega \; . \tag{5}$$

Multiplying (5) with $\overline{\mathbf{m}}^{n+1/2}$ we find

$$|\mathbf{m}^{n+1}|^2 - |\mathbf{m}^n|^2 = 0 \quad \Rightarrow \quad |\mathbf{m}^n| = |\mathbf{m}_0| \quad \forall n \; , \tag{6}$$

which means that the conservation of modulus carries over to the semi-discrete problem (5). Further,

$$\begin{aligned} \mathcal{E}(\mathbf{m}^{n+1}) - \mathcal{E}(\mathbf{m}^n) &= \tfrac{1}{2} \int_\Omega (\nabla \mathbf{m}^{n+1} + \nabla \mathbf{m}^n) : (\nabla \mathbf{m}^{n+1} - \nabla \mathbf{m}^n) \, \mathrm{d}\mathbf{x} \\ &= -k \int_\Omega \Delta \overline{\mathbf{m}}^{n+1/2} \cdot \delta_t \mathbf{m}^{n+1/2} \\ &= -k \int_\Omega \Delta \overline{\mathbf{m}}^{n+1/2} \cdot \Big(\overline{\mathbf{m}}^{n+1/2} \times (\Delta \overline{\mathbf{m}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}) \Big) \, \mathrm{d}\mathbf{x} \\ &= -k \int_\Omega |\Delta \overline{\mathbf{m}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}|^2 \, \mathrm{d}\mathbf{x} \; . \end{aligned} \tag{7}$$

3.2 Mixed variational formulation

Let us focus on the problem to be solved in each timestep of (5): introducing new unknown $\mathbf{j} := \nabla \overline{\mathbf{m}}^{n+1/2}$ we end up with

$$\begin{aligned} \delta_t \mathbf{m}^{n+1/2} &= \overline{\mathbf{m}}^{n+1/2} \times (\operatorname{div} \overline{\mathbf{j}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}) \; , \\ \mathbf{j}^n &= \nabla \overline{\mathbf{m}}^n \; . \end{aligned}$$

These equations can be cast in weak form: seek $\mathbf{m}^{n+1} \in (L^2(\Omega))^3 \cap L^\infty(\Omega)^3$ and $\mathbf{j}^{n+1} \in (\boldsymbol{H}_0(\operatorname{div}; \Omega))^3$ such that

$$\begin{aligned} \left(\delta_t \mathbf{m}^{n+1/2}, \mathbf{v}\right)_0 &= \left(\operatorname{div} \overline{\mathbf{j}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}, \mathbf{v} \times \overline{\mathbf{m}}^{n+1/2}\right)_0 && \forall \mathbf{v} \in (L^2(\Omega))^3 \; , \\ \left(\mathbf{j}^{n+1}, \mathbf{q}\right)_0 &= -\left(\operatorname{div} \mathbf{q}, \mathbf{m}^{n+1}\right)_0 && \forall \mathbf{q} \in (\boldsymbol{H}_0(\operatorname{div}; \Omega))^3 \; . \end{aligned} \tag{8}$$

Here $(\cdot, \cdot)_0$ stands for the $L^2(\Omega)$ inner product.

Now, let us consider an abstract spatial Galerkin discretization based on the finite-dimensional spaces $Q_h \subset (L^2(\Omega))^3$, $\mathbf{v}_h \in (\boldsymbol{H}_0(\mathrm{div};\Omega))^3$: seek $\mathbf{m}_h^{n+1} \in Q_h$, $\mathbf{j}_h^{n+1} \in \mathbf{V}_h$ such that

$$\begin{aligned} \left(\delta_t \mathbf{m}_h^{n+1/2}, \mathbf{v}_h\right)_0 &= \left(\mathrm{div}\, \bar{\mathbf{j}}^{n+1/2} \times \overline{\mathbf{m}}_h^{n+1/2}, \mathbf{v} \times \overline{\mathbf{m}}_h^{n+1/2}\right)_0 \qquad \forall \mathbf{v}_h \in Q_h \ , \\ \left(\mathrm{div}\, \mathbf{q}_h, \mathbf{m}_h^{n+1}\right)_0 &= -\left(\mathbf{j}_h^{n+1}, \mathbf{q}_h\right)_0 \qquad \forall \mathbf{q}_h \in \mathbf{V}_h \ . \end{aligned} \tag{9}$$

The discrete energy at time t_n is given by

$$\mathcal{E}_n := \tfrac{1}{2} \int_\Omega |\mathbf{j}_h^n|^2 \, \mathrm{d}\mathbf{x} \ . \tag{10}$$

It decays according to

$$\begin{aligned} \mathcal{E}_{n+1} - \mathcal{E}_n &= \tfrac{1}{2} \int_\Omega |\mathbf{j}_h^{n+1}|^2 - |\mathbf{j}_h^n|^2 \, \mathrm{d}\mathbf{x} = \tfrac{1}{2} \left(\mathbf{j}_h^{n+1} + \mathbf{j}_h^n, \mathbf{j}_h^{n+1} - \mathbf{j}_h^n\right)_0 \\ &= \left(\mathbf{j}_h^{n+1} - \mathbf{j}_h^n, \bar{\mathbf{j}}_h^{n+1/2}\right)_0 = -k \left(\mathrm{div}\, \bar{\mathbf{j}}_h^{n+1/2}, \delta_t \mathbf{m}^{n+1/2}\right)_0 \\ &= -k \left\| \mathrm{div}\, \bar{\mathbf{j}}_h^{n+1/2} \times \overline{\mathbf{m}}_h^{n+1/2} \right\|_{L^2(\Omega)}^2 \ . \end{aligned}$$

So, regardless of the Galerkin spaces chosen, we obtain a stable method.

Remark. Recall the identity

$$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = \mathbf{b}(\mathbf{a} \cdot \mathbf{c}) - \mathbf{c}(\mathbf{a} \cdot \mathbf{b}) \quad \forall \mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3 \ . \tag{11}$$

This implies

$$\mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m}) = \Delta \mathbf{m} |\mathbf{m}|^2 - \mathbf{m}(\Delta \mathbf{m} \cdot \mathbf{m}) \ . \tag{12}$$

We thus may recast (9) as

$$\begin{aligned} \left(\delta_t \mathbf{m}_h^{n+1/2}, \mathbf{v}_h\right)_0 &= \left(\mathrm{div}\, \bar{\mathbf{j}}_h^{n+1/2} \cdot \mathbf{v}_h, |\overline{\mathbf{m}}_h^{n+1/2}|^2\right)_0 - \left(\mathrm{div}\, \bar{\mathbf{j}}_h^{n+1/2} \cdot \overline{\mathbf{m}}_h^{n+1/2}, \overline{\mathbf{m}}_h^{n+1/2} \cdot \mathbf{v}_h\right)_0 \quad \forall \mathbf{v}_h \in Q_h \\ \left(\mathrm{div}\, \mathbf{q}_h, \mathbf{m}_h^{n+1}\right)_0 &= -\left(\mathbf{j}_h^{n+1}, \mathbf{q}_h\right)_0 \qquad \forall \mathbf{q}_h \in \mathbf{V}_h \end{aligned} \tag{13}$$

Remark. The rationale behind the introduction of the mixed formulation (8) is to relax the demands on the regularity of the unknown function $\mathbf{m}$. In the eventual mixed problem we merely seek $\mathbf{m}$ in $(L^2(\Omega))^3$ so that discretization by means of discontinuous finite elements is feasible. This turns out to be crucial for the preservation of the modulus of $\mathbf{m}$.

3.3 Finite element Galerkin discretization

We assume that Ω is covered by a triangular mesh $\mathcal{M}$. The following conforming finite element trial spaces will be employed:

for $(L^2(\Omega))^3$: space of $\mathcal{M}$-piecewise constant vectorfields on Ω,
for $\boldsymbol{H}(\mathrm{div};\Omega)$: lowest order Raviart-Thomas finite element space

The local shape functions for the Raviart-Thomas finite element space on a triangle with vertices $\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3$ are given by

$$\mathbf{b}_e(\mathbf{x}) = \frac{|e|}{2|T|}(\mathbf{x} - \mathbf{a}^e) \quad , \quad e = 1, 2, 3 \tag{14}$$

where $|e|$ is the length of edge opposite to the vertex $\mathbf{a}^e$. If $\mathbf{n}_e$ designates the exterior unit normal vector to that edge, these local shape functions are dual to the local degrees of freedom:

$$\int_e \mathbf{b}_i \cdot \mathbf{n}_e \, \mathrm{d}S = |e|\delta_{ie} \quad ; \quad i, e = 1, 2, 3 \, . \tag{15}$$

By fixing global degrees of freedom of the form (15) we can enforce the *tangential continuity* of the piecewise polynomial vector-fields contained in the finite element space. This amounts to conformity in $\boldsymbol{H}(\mathrm{div}; \Omega)$.

The key observation is that the choice of Q_h allows a local testing of the first equation in (9). In particular, we may choose $\mathbf{v}_h := \chi_K \overline{\mathbf{m}}_h^{n+1/2}$, χ_K the characteristic function of a cell K. This converts the first equation of (9) to

$$\frac{1}{2\tau}(\mathbf{m}_{h|K}^{n+1} - \mathbf{m}_{h|K}^{n})(\mathbf{m}_{h|K}^{n+1} + \mathbf{m}_{h|K}^{n}) = \frac{1}{2\tau}|\mathbf{m}_{h|K}^{n+1}|^2 - |\mathbf{m}_{h|K}^{n}|^2 = 0 \, . \tag{16}$$

Obviously, the norm of $\mathbf{m}_h$ is strictly conserved during timestepping.

3.4 Mesh

We choose $\Omega = [0, 1]^2$ as our computational domain and cover it by a regular mesh of triangles of width h.

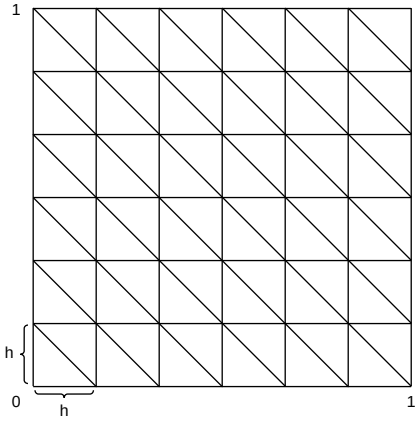


Figure 1: computational domain

With the choice of the Mesh $\mathcal{M}$ we can now write down our local shape functions. With $\mathbf{m} \in Q_h$ and Q_h discretized with the space of $\mathcal{M}$-piecewise constant vector fields, we write $\mathbf{m} = \sum_{k,i} m_{ki} \mathbf{b}_{ki}$, where m_{ki} is the coefficient and $\mathbf{b}_{ki}$ the local shape function on triangle k of the i-component of the vector field. Obviously

$$\mathbf{b}_{ki}(\mathbf{x}) = \begin{cases} \vec{e}_i & \mathbf{x} \in \Omega_k \\ 0 & elsewhere \end{cases} \, . \tag{17}$$

Likewise with $\mathbf{j} \in V_h$ and V_h discretized with the lowest order Raviart-Thomas finite element space, we write $\mathbf{j} = \sum_{e,i} j_{ei}\mathbf{B}_{ei}$, as a sum over all edges e and components i. Because we use a 2D vector to describe each component of our 3D vector field, we end up with a 2×3 tensor for $\mathbf{B}_{ei}$. For our regular mesh we can now write down the three different kinds of local shape functions: If e is a vertical edge of Ω_k

$$\mathbf{B}_{ei}(\mathbf{x}) = \begin{cases} \frac{s_k}{h}\binom{x-x_0-h}{y-y_0} \otimes \vec{e}_i & \mathbf{x} \in \Omega_k \\ 0 & elsewhere \end{cases}, \tag{18}$$

with (x_0, y_0) being the position of the lower left vertex of the square holding Ω_k. We introduced s_k which we define as 1 if Ω_k is a lower triangle and -1 if Ω_k is an upper triangle. The introduction of s_k ensures the continuity of the normal component of $\mathbf{B}$ between neighboring triangles. If e is a horizontal edge of Ω_k

$$\mathbf{B}_{ei}(\mathbf{x}) = \begin{cases} \frac{s_k}{h}\binom{x-x_0}{y-y_0-h} \otimes \vec{e}_i & \mathbf{x} \in \Omega_k \\ 0 & elsewhere \end{cases}, \tag{19}$$

and if e is a diagonal edge of Ω_k, then

$$\mathbf{B}_{ei}(\mathbf{x}) = \begin{cases} \frac{\sqrt{2}}{h}\binom{x-x_0}{y-y_0} \otimes \vec{e}_i & \mathbf{x} \in \Omega_k,\ \Omega_k \text{ lower triangle} \\ \frac{\sqrt{2}}{h}\binom{x_0+h-x}{y_0+h-y} \otimes \vec{e}_i & \mathbf{x} \in \Omega_k,\ \Omega_k \text{ lower triangle} \\ 0 & elsewhere \end{cases}. \tag{20}$$

Note that the divergence of of lowest order Raviart-Thomas local shape functions is always constant. On our mesh we get

$$\operatorname{div} \mathbf{B}_{ei}(\mathbf{x}) = \begin{cases} s_k \frac{|e|}{|\Omega_k|}\vec{e}_i & \mathbf{x} \in \Omega_k \\ 0 & \text{elsewhere} \end{cases};$$

$$\frac{|e|}{|\Omega_k|} = \begin{cases} \frac{2}{h} & e \text{ horizontal or vertical} \\ \frac{2\sqrt{2}}{h} & e \text{ diagonal} \end{cases}.$$

If we now use these local shape functions to describe our unknown vector field $\mathbf{m}$ and its gradient $\mathbf{j} := \nabla\mathbf{m}$ on a mesh of mesh width $h = 1/N$, we can describe our solution with $15N^2 + 6N$ coefficients, $6N^2$ of which describing $\mathbf{m}$ and the rest describing $\mathbf{j}$.

3.5 Nonlinear system of equations

In this section we fully discretize our so far time discrete system (13) with the finite element space described in section 3.3 i.e. the local shape functions derived in the previous section. This will lead us to a nonlinear system of equations which we can write down in a vector form $\mathbf{A}(\mathbf{x}) = 0$, where $\mathbf{x}$ holds the unknown coefficients of the finite element basis functions. Later we will solve this system for every timestep using a Newton Iteration.

Let us recall the time discrete system (13)

$$\begin{aligned} \left(\delta_t \mathbf{m}_h^{n+1/2}, \mathbf{v}_h\right)_0 - \left(\operatorname{div} \bar{\mathbf{j}}_h^{n+1/2} \cdot \mathbf{v}_h, |\overline{\mathbf{m}}_h^{n+1/2}|^2\right)_0 + \left(\operatorname{div} \bar{\mathbf{j}}_h^{n+1/2} \cdot \overline{\mathbf{m}}_h^{n+1/2}, \overline{\mathbf{m}}_h^{n+1/2} \cdot \mathbf{v}_h\right)_0 &= 0 \quad \forall \mathbf{v}_h \in Q_h \\ \left(\operatorname{div} \mathbf{q}_h, \mathbf{m}_h^{n+1}\right)_0 + \left(\mathbf{j}_h^{n+1}, \mathbf{q}_h\right)_0 &= 0 \quad \forall \mathbf{q}_h \in \mathbf{V}_h \end{aligned}$$

With $\mathbf{v}_h = \sum_{k,i} v_{ki}\mathbf{b}_{ki}$ and $\mathbf{q}_h = \sum_{k,e,i} q_{kei}\mathbf{B}_{kei}$ and since (13) is linear in $\mathbf{v}$ and $\mathbf{q}$ respectively, demanding (13) for all $\mathbf{v}_h \in Q_h$ and for all $\mathbf{q}_h \in \mathbf{V}_h$ is equivalent to demanding

$$\begin{aligned}\left(\delta_t \mathbf{m}_h^{n+1/2}, \mathbf{b}_{ki}\right)_0 - \left(\operatorname{div} \bar{\mathbf{j}}^{n+1/2} \cdot \mathbf{b}_{ki}, |\overline{\mathbf{m}}_h^{n+1/2}|^2\right)_0 + \left(\operatorname{div} \bar{\mathbf{j}}^{n+1/2} \cdot \overline{\mathbf{m}}_h^{n+1/2}, \overline{\mathbf{m}}_h^{n+1/2} \cdot \mathbf{b}_{ki}\right)_0 &= 0 \quad \forall k,i \\ \left(\operatorname{div} \mathbf{B}_{ei}, \mathbf{m}_h^{n+1}\right)_0 + \left(\mathbf{j}_h^{n+1}, \mathbf{B}_{ei}\right)_0 &= 0 \quad \forall e,i \end{aligned} \tag{21}$$

We now replace $\mathbf{m}_h$ with $\sum_{k,i} m_{ki}\mathbf{b}_{ki}$ and $\mathbf{j}_{ei}$ with $\sum_{e,i} j_{ei}\mathbf{B}_{ei}$ and simplify each of the terms of (21) separately. Because of the local support of the $\mathbf{b}_{ki}$'s it is sufficient to integrate the first line over Ω_k only. For the first term we get

$$\begin{aligned}&\left(\delta_t \overline{\mathbf{m}}_h^{n+1/2}, \mathbf{b}_{ki}\right)_0 \\ =&\int_{\Omega_k} \delta_t \overline{m}_{ki}^{n+1/2} \mathbf{b}_{ki} \cdot \mathbf{b}_{ki} \mathrm{d}\Omega \\ =&\delta_t \overline{m}_{ki}^{n+1/2} \int_{\Omega_k} \mathbf{b}_{ki} \cdot \mathbf{b}_{ki} \mathrm{d}\Omega \\ =&\frac{h^2}{2} \delta_t \overline{m}_{ki}^{n+1/2} .\end{aligned}$$

Likewise the second term becomes

$$\begin{aligned}&-\left(\operatorname{div} \bar{\mathbf{j}}^{n+1/2} \cdot \mathbf{b}_{ki}, |\overline{\mathbf{m}}_h^{n+1/2}|^2\right)_0 \\ =&-\int_{\Omega_k} \operatorname{div}\Big(\sum_{e \in E_k} \bar{j}_{ei}^{n+1/2} \mathbf{B}_{ei}\Big) \cdot \mathbf{b}_{ki} \cdot \sum_{l=1}^{3} \big(\overline{m}_{kl}^{n+1/2} \mathbf{b}_{kl}\big)^2 \mathrm{d}\Omega \\ =&-\sum_{e \in E_k} \bar{j}_{ei}^{n+1/2} \cdot \sum_{l=1}^{3} \big|\overline{\mathbf{m}}_h^{n+1/2}\big|^2 \int_{\Omega_k} (\operatorname{div} \mathbf{B}_{ei} \cdot \mathbf{b}_{ki}) \cdot (\mathbf{b}_{kl} \cdot \mathbf{b}_{kl}) \mathrm{d}\Omega \\ =&-s_k \sum_{e \in E_k} |e| \bar{j}_{ei}^{n+1/2} \cdot \sum_{l=1}^{3} \big|\overline{m}_{kl}^{n+1/2}\big|^2 .\end{aligned}$$

Again, because of the local support of $\mathbf{b}_{ki}$ we only sum over the Raviart-Thomas shape functions belonging to edges of cell k, $e \in E_k$. Similarly the third term can be written as

$$\begin{aligned}&\left(\operatorname{div} \bar{\mathbf{j}}^{n+1/2} \cdot \overline{\mathbf{m}}_h^{n+1/2}, \overline{\mathbf{m}}_h^{n+1/2} \cdot \mathbf{b}_{ki}\right)_0 \\ =&\int_{\Omega_k} \sum_{e \in E_k} \Big(\sum_{l=1}^{3} \operatorname{div}(\bar{j}_{el}^{n+1/2} \mathbf{B}_{el}) \cdot \overline{m}_{kl}^{n+1/2} \mathbf{b}_{kl}\Big) \cdot (\overline{m}_{ki}^{n+1/2} \mathbf{b}_{ki} \cdot \mathbf{b}_{ki}) \mathrm{d}\Omega \\ =&\sum_{e \in E_k} \sum_{l=1}^{3} \bar{j}_{el}^{n+1/2} \overline{m}_{kl}^{n+1/2} \overline{m}_{ki}^{n+1/2} \int_{\Omega_k} (\operatorname{div} \mathbf{B}_{el} \cdot \mathbf{b}_{kl}) \cdot (\mathbf{b}_{ki} \cdot \mathbf{b}_{ki}) \mathrm{d}\Omega \\ =&\sum_{e \in E_k} \sum_{l=1}^{3} s_k |e| \bar{j}_{el} \overline{m}_{kl}^{n+1/2} \overline{m}_{ki}^{n+1/2} .\end{aligned}$$

Now we look at the second line of (21). Because a local shape function belonging to edge e has a support in both neighboring cells $k|e \in E_k$, we integrate only over these two triangles. The first term becomes

$$\begin{aligned}
&\left(\operatorname{div} \mathbf{B}_{ei}, \mathbf{m}_h^{n+1}\right)_0 \\
&= \sum_{k|e\in E_k} \int_{\Omega_k} \operatorname{div} \mathbf{B}_{ei} \cdot m_{ki}^{n+1} \mathbf{b}_{ik} \mathrm{d}\Omega \\
&= \sum_{k|e\in E_k} m_{ki}^{n+1} \int_{\Omega_k} \operatorname{div} \mathbf{B}_{ei} \cdot \mathbf{b}_{ik} \mathrm{d}\Omega \\
&= \sum_{k|e\in E_k} s_k |e| m_{ki}^{n+1} \ ,
\end{aligned}$$

and the second term can be written as

$$\begin{aligned}
&\left(\mathbf{j}_h^{n+1}, \mathbf{B}_{ei}\right)_0 \\
&= \sum_{k|e\in E_k} \int_{\Omega_k} \sum_{e'\in E_k} (j_{e'i}^{n+1} \mathbf{B}_{e'i}) \cdot \mathbf{B}_{ei} \mathrm{d}\Omega \\
&= \sum_{k|e\in E_k} \sum_{e'\in E_k} j_{e'i}^{n+1} \int_{\Omega_k} \mathbf{B}_{e'i} : \mathbf{B}_{ei} \mathrm{d}\Omega \ .
\end{aligned}$$

The Integral $I_{ee'} := \int_{\Omega_k} \mathbf{B}_{e'i} : \mathbf{B}_{ei} \mathrm{d}\Omega$ can be directly calculated with our local shape functions obtained in the last section, it evaluates to $\frac{h^2}{3}$ if $e' = e$, $-\frac{h^2}{6}$ if one of e and e' is a horizontal and the other a vertical edge and 0 in any other case. Now we can write down our fully discretized system of equations.

$$\mathbf{A}(\mathbf{x}^{n+1}, \mathbf{x}^n) = 0 \quad ; \quad \mathbf{x}^{n+1} = \begin{pmatrix} m_{ki} \\ j_{ei} \end{pmatrix}^{n+1} , \tag{22}$$

$$\mathbf{A}(\mathbf{x}^{n+1}, \mathbf{x}^n) = \begin{pmatrix} \frac{h^2}{2} \delta_t \overline{m}_{ki}^{n+1/2} - s_k \sum\limits_{e\in E_k} |e| \overline{j}_{ei}^{n+1/2} \cdot \sum\limits_{l=1}^{3} |\overline{\mathbf{m}}_h^{n+1/2}|^2 + s_k \sum\limits_{l=1}^{3} |e| \overline{j}_{el}^{n+1/2} \overline{m}_{kl}^{n+1/2} \overline{m}_{ki}^{n+1/2} \\ \sum\limits_{k|e\in E_k} \left(s_k |e| m_{ki}^{n+1} + \sum\limits_{e'\in E_k} j_{e'i}^{n+1} I_{ee'} \right) \end{pmatrix} \tag{23}$$

3.6 Boundary conditions

Since the coefficients of the Raviart-Thomas local shape functions directly determine the normal component of $\mathbf{j}$ over the edges of the triangles, we can easily satisfy our boundary condition

$$\frac{\partial \mathbf{m}}{\partial \mathbf{n}} = \mathbf{j} \cdot \mathbf{n} = 0 \quad \text{on } \partial\Omega \ ,$$

if all coefficients belonging to edges on $\partial\Omega$ are zero at all times. Thus these $12N$ coefficients are no longer degrees of freedom and can be eliminated from the global solution vector $\mathbf{x}$, which leaves us with $15N^2 - 6N$ unknowns.

4 Solution of the discrete system

4.1 Linearization

We now know that our simulation must satisfy the fully discrete system of equations $\mathbf{A}(\mathbf{x}^{n+1}, \mathbf{x}^n) = 0$ for every time step. To perform a time step, the task of our program is solving this non linear system of equations i.e. from a given $\mathbf{x}^n$ finding $\mathbf{x}^{n+1}$. We will solve $\mathbf{A}(\mathbf{x}^{n+1}, \mathbf{x}^n) = 0$ using Newtons Method, for this purpose we first need to find the Jacobian $\mathbf{DA} := \frac{\partial \mathbf{A}_i}{\partial \mathbf{x}_j^{n+1}}$

Using the calculations of $\mathbf{A}$ from the last section we get the following entries of $\mathbf{DA}$:

$$\frac{\partial \mathbf{A}_{ki}}{\partial m_{k'i'}^{n+1}} = \begin{cases} \frac{h^2}{2\kappa} + \frac{s_k}{2} \sum_{\substack{l=1 \\ l \neq i}}^{3} \sum_{e \in E_k} |e| \overline{j}_{el}^{n+1/2} \overline{m}_{kl}^{n+1/2} & k'=k, \\ & i'=i \\ -s_k \sum_{e \in E_k} |e| \overline{j}_{ei}^{n+1/2} \overline{m}_{ki'}^{n+1/2} + \frac{s_k}{2} \sum_{e \in E_k} |e| \overline{j}_{ei'}^{n+1/2} \overline{m}_{ki}^{n+1/2} & k'=k, \\ & i' \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial \mathbf{A}_{ki}}{\partial j_{e'i'}^{n+1}} = \begin{cases} -\frac{s_k}{2} |e'| \sum_{\substack{l=1 \\ l \neq i}}^{3} \left| \overline{m}_{kl}^{n+1/2} \right|^2 & e' \in E_k, \\ & i'=i \\ -\frac{s_k}{2} |e'| \overline{m}_{ki}^{n+1/2} \overline{m}_{ki'}^{n+1/2} & e' \in E_k, \\ & i' \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial \mathbf{A}_{ei}}{\partial j_{e'i'}^{n+1}} = \begin{cases} \frac{2}{3} h^2 & e'=e, \\ & i'=i \\ -\frac{1}{6} h^2 & \exists k | e \in H_k, e' \in V_k \text{ or } e \in V_k, e' \in H_k, \\ & i'=i \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\partial \mathbf{A}_{ei}}{\partial m_{k'i'}^{n+1}} = \begin{cases} s_k |e| & i'=i, \\ & e \in \Omega_k \\ 0 & \text{otherwise} \end{cases}$$

Obviously $\mathbf{DA}$ is very sparse. It has 12 non-zero entries for each of the $6N^2$ lines corresponding to coefficients of $\mathbf{m}$, and 4 or 5 non-zero entries for the lines corresponding to $\mathbf{j}$ coefficients, 4 for diagonal edges and 5 for the other ones.

Because the mesh does not move with time, the structure of $\mathbf{DA}$ stays constant during the whole simulation. Even more, most of entries of $\mathbf{DA}$ stay constant and only depend on the mesh width h. These properties can be used to considerably speed up the assembly of $\mathbf{DA}$ in our program code.

4.2 Newton Iteration

At a given time our program must assemble $\mathbf{A}$ and $\mathbf{DA}$ to then find the solution of the next time by solving $\mathbf{A}(\mathbf{x}^{n+1}, \mathbf{x}^n) = 0$. One Newton step is done by iterating the iterated solution $\mathbf{x}^i$

$$\mathbf{x}^{i+1} = \mathbf{x}^i - \mathbf{DA}(\mathbf{x}^i, \mathbf{x}^n) \backslash \mathbf{A}(\mathbf{x}^i, \mathbf{x}^n) \ .$$

As initial guess we can use the solution of the last timestep.

$$\mathbf{x}^0 = \mathbf{x}^n \ .$$

Besides computing **A** and assembling **DA**, finding the solution of the linear system of equations **DA****A** for every Newton step is the crucial part of the simulation. In our first MATLAB implementation of the simulation we simply use the MATLAB \-operator.

5 Convergence

5.1 Initial condition and analytical solution

In order to measure the discretization error of the numerical method we need to find an exact solution of our governing equations (1) for given initial conditions. We can then run the simulation with the same initial conditions and compare the results of the simulation with the exact solution to measure the error.

We have to find an analytical solution of our system of equation (1), we know it must satisfy both $|\mathbf{m}| = 1$ and the boundary condition $\frac{\partial \mathbf{m}}{\partial \mathbf{n}} = 0$ at all times. We intuitively choose

$$\mathbf{m}_f = \begin{pmatrix} \cos(e^{-\lambda t}\cos(\pi x)\cos(\pi y)) \\ \sin(e^{-\lambda t}\cos(\pi x)\cos(\pi y)) \\ 0 \end{pmatrix};. \tag{24}$$

One can use Mathematica to verify that this indeed is a solution of (1) if simply

$$\lambda = 2\pi^2 \ .$$

Thus, to check the convergence of our simulation we will use the initial condition

$$\mathbf{m}_0 = \begin{pmatrix} \cos(\cos(\pi x)\cos(\pi y)) \\ \sin(\cos(\pi x)\cos(\pi y)) \\ 0 \end{pmatrix} \ . \tag{25}$$

5.2 The forcing term

When we started working on this term project, we did not know that there is a simple analytical solution of (1). Since we still wanted to measure the convergence rate somehow, we forced the simulation to follow a given analytical solution by adding a so called forcing term. This procedure was quite instructive we will shortly explain it here although in the end non of this could be used in the simulation and is more of theoretical concern.

First of all we choose a analytical function $\mathbf{m}_f(\mathbf{x}, t)$ we want our simulation to follow. Then, to force our simulation to follow $\mathbf{m}_f$, our governing equations are modified by adding the forcing term $\mathbf{f}(\mathbf{x}, t)$, ie. by rewriting (1) as

$$\begin{aligned} \frac{\partial \mathbf{m}}{\partial t} &= \mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m}) + \mathbf{f}(\mathbf{x}, t) \quad \text{in }]0, T[\times\Omega \ , \\ \mathbf{m}(0) &= \mathbf{m}_0 \quad \text{in } \Omega \ , \\ \frac{\partial \mathbf{m}}{\partial \mathbf{n}} &= 0 \quad \text{on }]0, T[\times\partial\Omega \ . \end{aligned} \tag{26}$$

$\mathbf{f}(\mathbf{x}, t)$ can directly be calculated when we assume $\mathbf{m} = \mathbf{m}_f$ in this equation. Because we introduce the forcing term, equation (4) also has to be modified to

$$\frac{d|\mathbf{m}|^2}{dt} = 2\mathbf{m} \cdot \frac{\partial \mathbf{m}}{\partial t} = 2\mathbf{m} \cdot (\mathbf{m} \times (\Delta \mathbf{m} \times \mathbf{m}) + \mathbf{f}) \; . \tag{27}$$

Note that the norm of $\mathbf{m}$ is only conserved if $\mathbf{f} \cdot \mathbf{m} = 0$ for all t. After temporal discretization with the method of Heun we get

$$\delta_t \mathbf{m}^{n+1/2} = \overline{\mathbf{m}}^{n+1/2} \times (\Delta \overline{\mathbf{m}}^{n+1/2} \times \overline{\mathbf{m}}^{n+1/2}) + \overline{\mathbf{f}}^{n+1/2}, \tag{28}$$

where $\overline{\mathbf{f}}^{n+1/2} = \frac{1}{2}\mathbf{f}(\mathbf{x}, t^{n+1}) + \mathbf{f}(\mathbf{x}, t^n)$. Multiplying (28) with $\overline{\mathbf{m}}^{n+1/2}$ yields

$$|\mathbf{m}^{n+1}|^2 - |\mathbf{m}^n|^2 = \frac{k}{4}(\mathbf{f}^{n+1} \cdot \mathbf{m}^n + \mathbf{f}^n \cdot \mathbf{m}^{n+1}) \; , \tag{29}$$

thus, in the time discrete form, the modulus of $\mathbf{m}$ is only conserved if the right hand side of equation (29) is zero. This means, that we can force our simulation to follow any given analytical function $\mathbf{m}_f$ as long as

$$\mathbf{f}^{n+1} \cdot \mathbf{m}^n + \mathbf{f}^n \cdot \mathbf{m}^{n+1} = 0 \; ,$$

but finding such a $\mathbf{m}$ can be as hard as finding a solution to (1).

Assuming we found such a $\mathbf{m}$, we could discretize the resulting forcing term in the same way as we discretized equation (13) and end up with an additional term contributing to $\mathbf{A}(\mathbf{x})$ of the form

$$\int_{\Omega_k} \mathbf{f}(\mathbf{x}, t) \mathrm{d}\Omega_k \; . \tag{30}$$

The integral (30) could be calculated over every cell of the mesh. In our simulation this could be done with a numerical quadrature formula which matches the expected convergence rate. Hence a quadrature rule of order 3 would be sufficient.

5.3 Error norms

Fortunately we know an exact analytical solution and we can directly examine the L^2 and H^1 error norms to measure the convergence rate of the simulation with

$$\begin{aligned} \|\mathbf{m}_{sol} - \mathbf{m}\|^2_{L^2(\Omega)} &= \int_\Omega (\mathbf{m}_{sol} - \mathbf{m})^2 \, \mathrm{d}\Omega \\ &= \sum_k \sum_i \int_{\Omega_k} (m_{sol,i} - m_{ki})^2 \, \mathrm{d}\Omega_k \end{aligned} \tag{31}$$

and

$$\begin{aligned} |\mathbf{m}_{sol} - \mathbf{m}|^2_{H^1(\Omega)} &= \int_\Omega (\nabla \mathbf{m}_{sol} - \nabla \mathbf{m})^2 \, \mathrm{d}\Omega \\ &= \sum_k \sum_i \int_{\Omega_k} \left(\nabla m_{sol,i} - \sum_{e \in E_k} j_{ei}\right)^2 \mathrm{d}\Omega_k. \end{aligned} \tag{32}$$

The integrals are computed over each cell with a quadrature rule. Since we use piecewise constant basis functions to discretize $\mathbf{m}$, our L^2 error will be dominated by linear terms. Likewise, since for $\nabla \mathbf{m}$ we use piecewise linear basis function, our H^1 error will be mainly quadratic, thus using a quadrature rule of order 3 to estimate both errors will be sufficiently accurate.

5.4 Simulation results

We can now run the simulation with the initial condition (25) and compute the error at given time with the analytical solution and the norms described in the previous sections.

We measured the convergence of the numerical scheme at four different times for h- and κ-refinement. The results obtained are plotted in figure 5.4 and 5.4. The errors that resulted from the simulation are plotted with circles and fitted by algebraic dotted curves. Note the algebraic convergence rates printed in the legends.

For the h-refinement we computed nice h^1 rates at all four times for both the L^2 and H^1 norm. Because we simulated with a very small time step, the error from the time discretization is not observable, even at relatively large N.

The results of the κ-refinement show a clear κ^2 dependence resulting from the second order method of Heun. For small κ the error of the time discretization starts deviating from the κ^2 because the $O(h^1)$ error of the spacial discretization starts dominating.

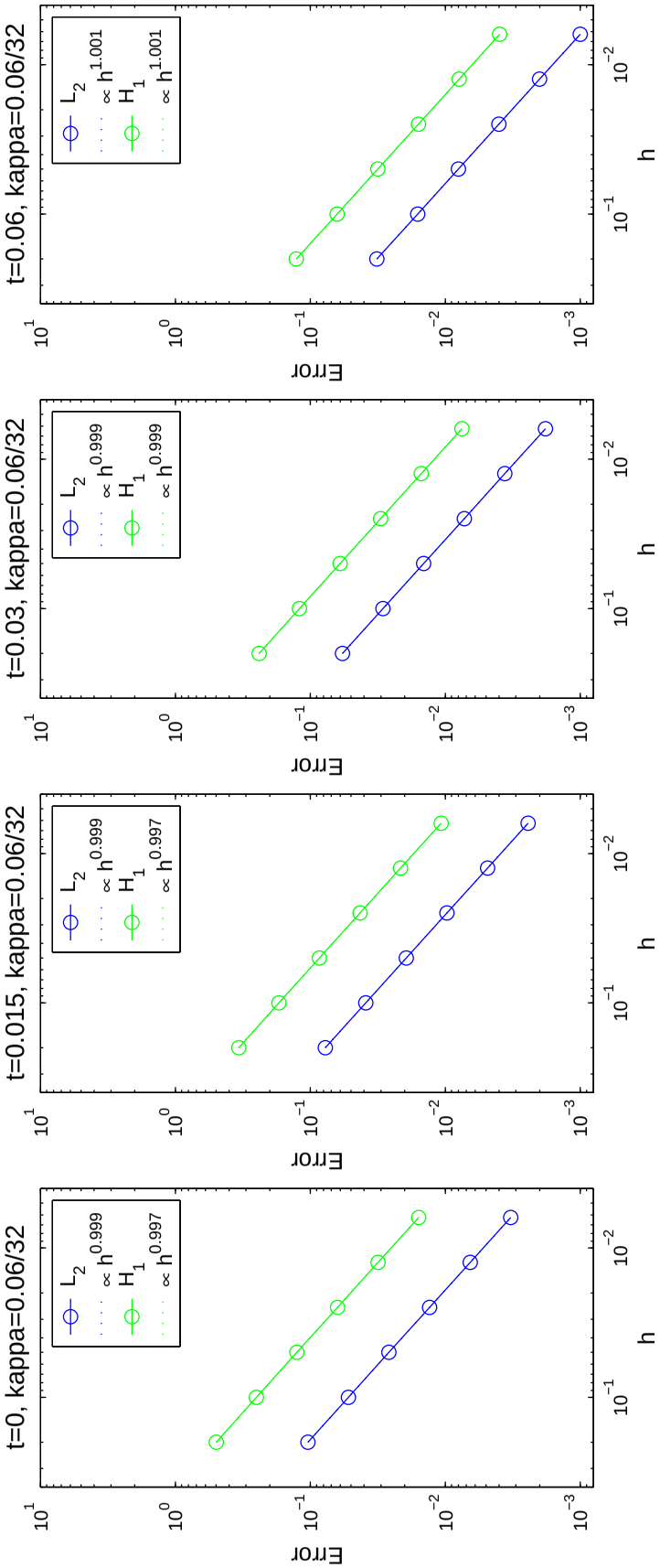


Figure 2: convergence for h-refinement

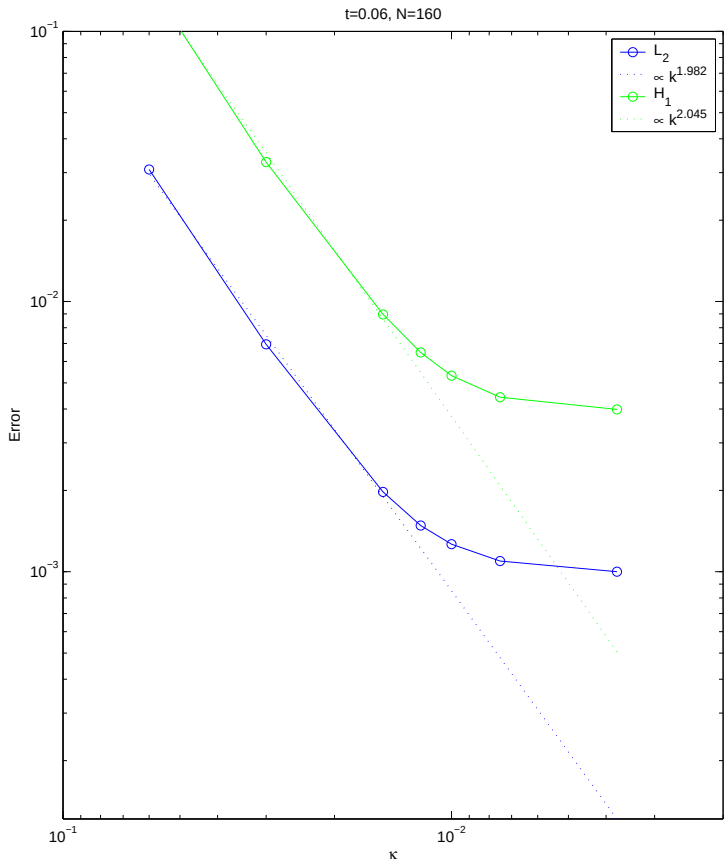


Figure 3: convergence for k-refinement

6 Parallelization

There are several ways to parallelize our simulation, we decided to implement two of them. The first one (section 6.1) simply uses a parallel solver, while the second one(section 6.2) splits the computational domain up into subdomains and parallelizes everything from assembling to solving.

We ran these two parallel versions of our simulation on ETH's Hreidar, which is a Beowulf class computing cluster. In the current version, it consists of 128 dual-CPU compute nodes, two login nodes and two file servers. Each compute node has two AMD Opteron 244 processors, 4 GB memory and ca. 100 GB disk storage.

6.1 SuperLU

When we rewrote our program in C, we found that in contrast to the MATLAB version, the assembly of **A** and **DA** only makes up a small part of the runtime and the program spends most of the time solving the system of equations. This acceleration of the assembling was made possible by utilizing the known structure of the Matrix **DA**. Therefore it was obvious to parallelize the solver. Owing to the ability to speedup a program on one node and also parallelize it on multiple nodes, we decided to use the SuperLU-library [2] for this purpose.

On a single node we gained a speedup around 2 only from using a more efficient way to assemble the matrix and the SuperLU-library instead of the MATLAB \-operator. The bottleneck of this implementation was the enormous need of memory. Despite of the column preordering before the LU-factorization there's still a remarkable fill-in, see table 6.1

N	entries in DA	entries in L+U	fill-in factor
20	44'000	440'000	10
40	180'000	2'800'000	15.5
80	700'000	15'000'000	21.5
160	2'800'000	80'000'000	28.5

Table 1: SuperLU fill-in

For this reason we were looking forward to see the results of the distributed version running on the Hreidar cluster. where we had around 2GB memory on each node. Since SuperLU distributes the storage of L and U, the memory problem seemed to be solved by increasing the amount of nodes. In fact the **DA** matrix is not distributed and also still computed sequentially but the memory requirements are vanishing and the nearly inappreciable costs of the assembling saves us the time for distributing the hole matrix. Additionally we expected a speedup while raising the number of cpus. But unfortunately exactly the opposite happened. Table 6.1 shows that using more than one node slowed the program down up to a factor over 10.

This unpleasant phenomenon can be explained by the Hreidar-network. It is simply to slow and it does not make any sense to compute huge matrices on Hreidar. So, we were not running out of memory anymore, but because we couldn't use Hreidar just for us, the

N	1 proc	4 proc	16 proc
10	0.24 sec	34.6 sec	35.6 sec
160	5360 sec	10200 sec	6200 sec

Table 2: SuperLU performance

job size was limited now by the runtime. Finally we could run simulations with $N = 320$, which was disappointing, knowing that $N = 160$ is already possible in MATLAB and additionally the computation is also slower on Hreidar.

6.2 Domain decomposition

The results form our first simple parallelization attempt made clear that a useful parallel version of our simulation must address two problems: It should reduce the memory required, i.e. ideally require only memory to store the solution vectors and the sparse Matrix **DA**, and it should reduce the amount of communication between the nodes.

6.2.1 Idea

The key idea to our second parallelization approach, which takes these two points into account, is to use the conjugate gradient squared method [3] without preconditioner to solve $\mathbf{DA}\backslash\mathbf{A}$. The advantage of this method is that it only uses the $\mathbf{DA} \cdot \mathbf{x}$ operation to iteratively find the solution of the linear system of equations posed at every newton step, thus, in contrast to a standard CG method, $\mathbf{DA}^T \cdot \mathbf{x}$ must not be calculated.

We can split up the computational domain into subdomains and distribute the unknowns corresponding to those subdomains on the different nodes of the network. Because of the local support of the FEM basis functions, the assembly of **A** and **DA** and the computation of $\mathbf{DA} \cdot \mathbf{x}$ can now be done locally, as long as we exchange the coefficients lying on the subdomain borders between the nodes.

6.2.2 Implementation

Once again we rewrote our program, this time we converted it into a C++ Version that uses MPI to parallelize Vector and Matrix Classes for **x** and **DA**. We then used the CGS algorithm implemented in the *Iterative Template Library* [1]. All we had to do is to implement the abstract interface that is used by ITL with our vector and matrix classes.

Each parallel vector holds the coefficients of its subdomain plus a copy of all border coefficients, while each parallel matrix holds the rows corresponding to the local coefficients only. Most operations needed by CGS like adding, subtracting and scaling vectors can be done purely local, the dot product can be done in a simple `MPI_Allreduce` statement. Finally, the assembly of **DA**, the computation of **A** and the $\mathbf{DA} \cdot \mathbf{x}$ operation are also done locally but need the copies of the border coefficients, so before these methods can be called, the border coefficients must be exchanged and updated.

6.2.3 Splitting up the domain

There are several ways to split up our domain into np subdomains, three of which are illustrated in figure 6.2.3. The first one and the one we actually implemented simply splits the domain into horizontal stripes, it is the most simple version to code, but produces most communication between nodes. Additional to the $15N^2/np$ local coefficients, each Vector then holds $9N$ copies of neighbors and before every $\mathbf{DA}\cdot\mathbf{x}$ operation $9N\cdot(np-1)$ unknowns must be exchanged. This amount of communication can be reduced to $18N \cdot (\sqrt{np}-1)$if we split both horizontally and vertically and to $24N \cdot (\sqrt{np/2}-1)$ with an additional diagonal split which would probably be the fastest but most complicated version.

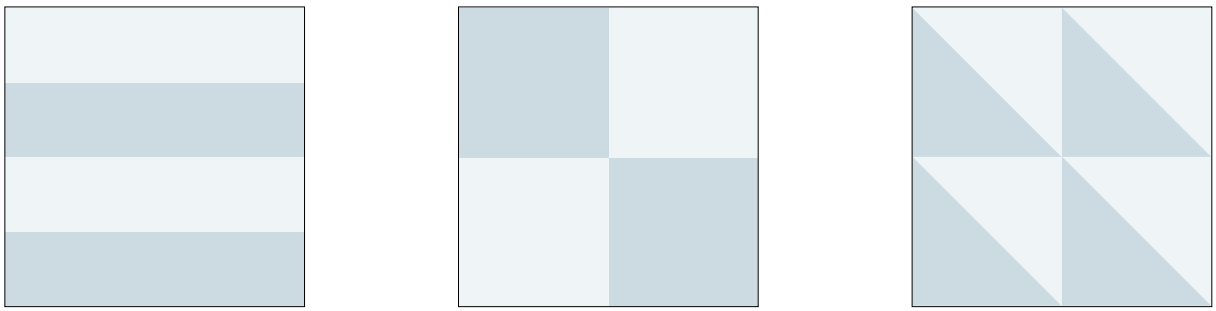
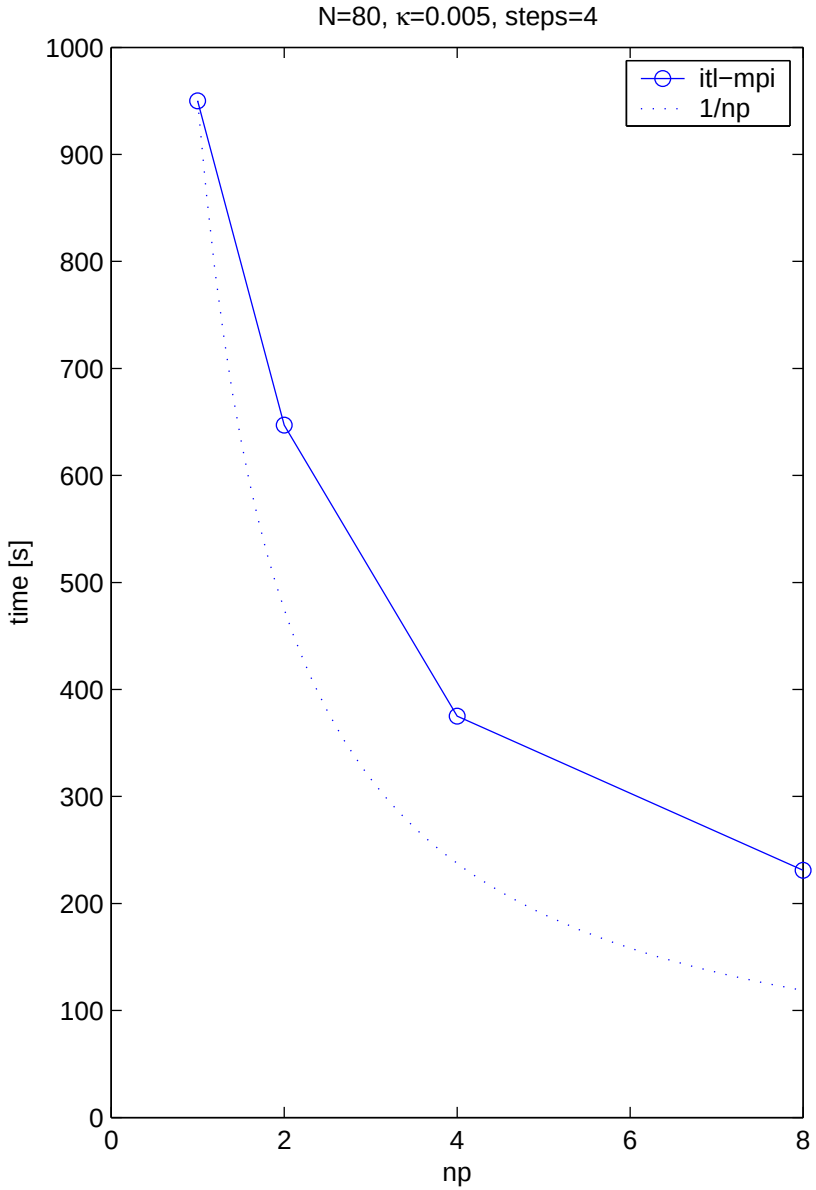


Figure 4: splitting up the domain

6.2.4 Performance results

We ran our domain decomposition program on the Hreidar cluster and compared the results with the SuperLU version. Even with our domain split only vertically, which produces an np instead of an $\sqrt{np}$ dependent amount of communication, our simulation scaled beautifully with increasing number of processors. The results are plotted in figure 6.2.4. The solid line is the $1/np$ curve for a perfect parallelization without communication. Obviously the ability to make use of knowing exactly how **DA** is assembled from the underlying FEM basis functions proved to be a huge advantage for our simulation, the SuperLU package of course does not know anything about **DA**.



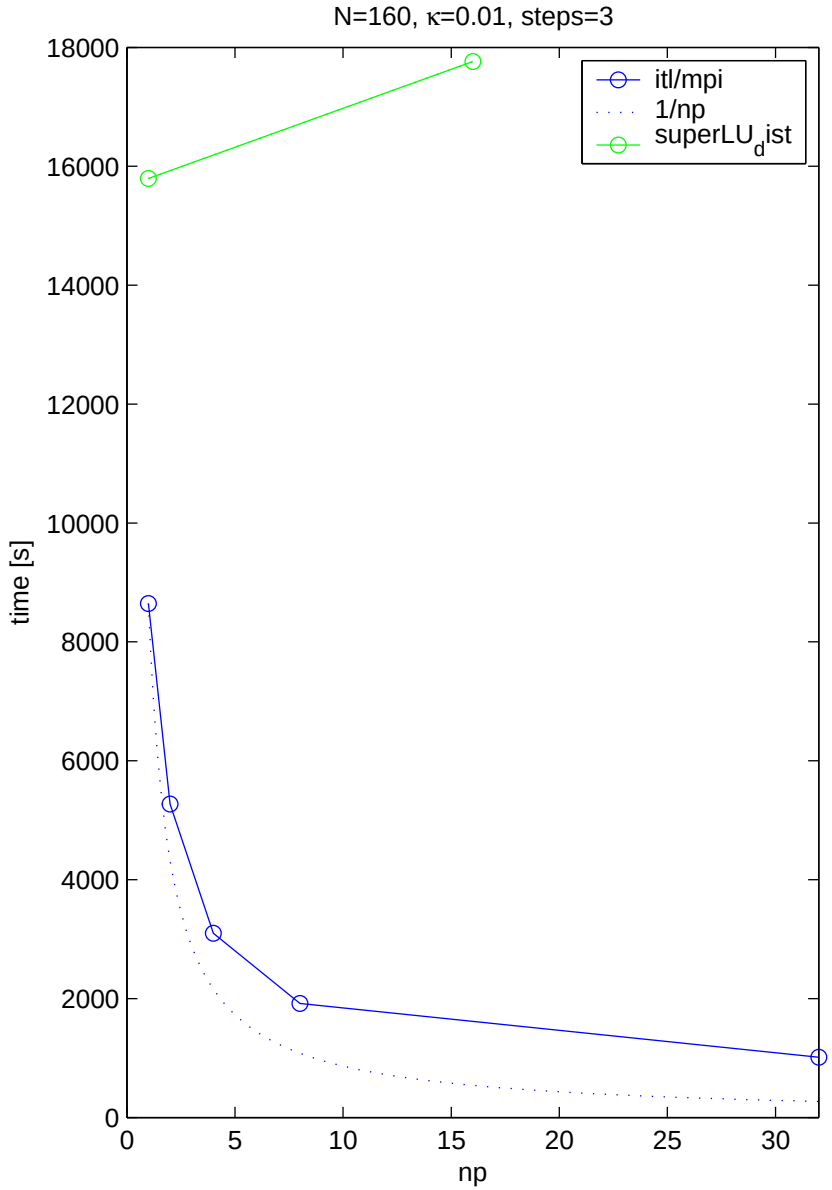


Figure 5: cpu time over increasing number of nodes np

A Program code

A.1 MATLAB

fem.m

```
function [mov,energy,err]=fem(N,t_end,Nt)
% Main method, runs simulation with specified parameters
%
%   N            system length and witdh
%   t_end        simulation time, run from t=0 until t_end
%   Nt           number of time steps to be performed

format compact;
format long;

h = 1/N;
kappa = t_end/Nt;

x = project_initial_condition(N);

time(1,1) = 0;
err(1,1) = get_err_abs_m(x);
%err(1,2) = get_err_l2(x,0); %use appropriate initial condition
%err(1,3) = get_err_h1(x,0); %use appropriate initial condition
energy(1,1) = get_energy(x);
plot_solution(x);
mov(1) = getframe;

for t=1:Nt
    disp('');
    disp(strcat('step'));
    disp(int2str(t));
    xNew = newton(kappa,h,x);
    if (xNew==0)
        disp('Newton Iteration did not converge')
        return
    end
    x = xNew;

    tn=t*kappa;
    time(t+1,1) = tn;
    err(t+1,1) = get_err_abs_m(x);
    %err(t+1,2) = get_err_l2(x,tn); %use appropriate initial condition
    %err(t+1,3) = get_err_h1(x,tn); %use appropriate initial condition
    energy(t+1,1) = get_energy(x);
    plot_solution(x);
    mov(t+1) = getframe;
end

format loose;
```

project_initial_conditions.m

```
function x0 = project_initial_condition(N)
% Projects the function specified by get_m_gradm_0 into FE space
% and returns the corresponding vector of coefficients

x0 = -ones(15*N*N+6*N,1);
h = 1/N;

% for each row and column
for j = 1:N
    for i = 1:N

        % set coordinates of the right-top of this to cells
        x = i*h;
        y = j*h;

        % get & save the m-vector from the center of the lower cell
        [m,gradm] = get_m_gradm_0(x-2*h/3,y-2*h/3);
        x0(6*N*(j-1) + 6*(i-1) + 1) = m(1);
        x0(6*N*(j-1) + 6*(i-1) + 2) = m(2);
        x0(6*N*(j-1) + 6*(i-1) + 3) = m(3);
        % get & save the m-vector from the center of the upper cell
        [m,gradm] = get_m_gradm_0(x-h/3,y-h/3);
        x0(6*N*(j-1) + 6*(i-1) + 4) = m(1);
        x0(6*N*(j-1) + 6*(i-1) + 5) = m(2);
        x0(6*N*(j-1) + 6*(i-1) + 6) = m(3);
        %get the j-matrix from the
        % middle of the left edge and save the -x - direction
        [m,gradm] = get_m_gradm_0(x-h,y-h/2);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 1) = -gradm(1,1);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 2) = -gradm(2,1);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 3) = -gradm(3,1);
        %get the j-matrix from the
        % middle of the bottom edge and save the -y - direction
        [m,gradm] = get_m_gradm_0(x-h/2,y-h);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 4) = -gradm(1,2);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 5) = -gradm(2,2);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 6) = -gradm(3,2);
        % Get the j-matrix from the middle of the right/top edge
        % and save the 1/sqrt(2)*(x+y) - direction
        [m,gradm] = get_m_gradm_0(x-h/2,y-h/2);
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 7) = 1/sqrt(2)*(gradm(1,1)+gradm(1,2));
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 8) = 1/sqrt(2)*(gradm(2,1)+gradm(2,2));
        x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 9) = 1/sqrt(2)*(gradm(3,1)+gradm(3,2));

        %if we are at right border
        if i==N
            [m,gradm] = get_m_gradm_0(x,y-h/2);
            x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 10) = -gradm(1,1);
            x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 11) = -gradm(2,1);
            x0(6*N*N + (j-1)*(9*N+3) + (i-1)*9 + 12) = -gradm(3,1);
        end
    end

    %if we are at top border
    if j==N
        for i=1:N
             % set coordinates of the right-top of this to cells
            x = i*h;
            y = j*h;
            [m,gradm] = get_m_gradm_0(x-h/2,y);
            x0(15*N*N + 3*N + (i-1)*3 + 1) = -gradm(1,2);
            x0(15*N*N + 3*N + (i-1)*3 + 2) = -gradm(2,2);
            x0(15*N*N + 3*N + (i-1)*3 + 3) = -gradm(3,2);
        end
    end
end
```

newton.m

```
function xi = newton(kappa,h,x0)
% Finds the solution of non-linear system of equations A(x)=0
% through newton iteration
%
%   kappa         time step width
%   h             cell width
%   x0            solution at previous timestep (n) which serves as initial
%                 guess for newton iteration
%   xi            solution at next timestep (n+1), newton iterated solution

L     = length(x0);
i     = 0;
imax  = 50;
tol   = 1e-6;
dx    = ones(L,1);
xi    = zeros(L,1);
normA = 1;

%while iteration has not converged to solution
while normA > tol

    %if iterations does not converge stop
    if i >= imax
        xi = 0;
        return
    end

    %update counter
    i = i+1;

    %calculate A and DA
    [A,DA]  = get_A(kappa,h,x0,xi);

    %do newton step
    dx = DA\A;
    xi = xi-dx;
    normA = sqrt(A'*A);
    disp(normA);
end
```

get_A.m

```
function [A,DA] = get_A(kappa,h,x0,xi)
% Assembles A(x0;xi) and the Jacobian DA from local contributions of A
% computed by get_Aloc
%
%   kappa        time step width
%   h            cell width
%   x0           solution at previous timestep (n) which also serves as
%                initial guess for newton iteration
%   xi           newton iterated solution

% set true to force boundary conditions
forcebound = true;

% calculate parameters
N      = 1/h;
nc     = 2*N*N;
nc3    = 3 * nc;
L      = length(x0);
A      = zeros(L,1);
entries = 72 * nc - 36 * N;
iarray = zeros(entries,1);
jarray = zeros(entries,1);
valuearray = zeros(entries,1);
arrayindex = 1;

% for all cells of mesh
for ic = 0:N-1
  for jc = 0:N-1
    for kc = 0:1

        % calculate index of current cell and indices of edges of current
        % cell in global system of equations. Set indices so that e1 points
        % to the diagonal edge
        k  = 6*(jc*N+ic)+3*kc+1;
        e1 = 6*N*N + (9*N+3)*jc + 9*ic + 7;
        if kc == 0
            e2 = 6*N*N + (9*N+3)*jc + 9*ic + 1;
            e3 = 6*N*N + (9*N+3)*jc + 9*ic + 4;
        else
            e2 = 6*N*N + (9*N+3)*jc + 9*(ic+1) + 1;
            if jc~=N-1
                e3 = 6*N*N + (9*N+3)*(jc+1) + 9*ic + 4;
            else
                e3 = 6*N*N + (9*N+3)*(jc+1) + 3*ic + 1;
            end
        end

        % check if e2 or e3 are indices of boundary basis functions
        e2isboundary = (ic==0 && kc == 0) || (ic==N-1 && kc==1);
        e3isboundary = (jc==0 && kc == 0) || (jc==N-1 && kc==1);

        % set sign of thomas raviart basis functions
        if kc == 0
            sk = 1;
        else
            sk = -1;
        end

        % calculate local A and DA
        [Aloc,DAloc]  = get_Aloc(kappa,h,k,e1,e2,e3,sk,x0,xi);

        % assemble global A from local contributions
        for i=0:2
            A(k+i)  = A(k+i) + Aloc(1+i);
        end
        for i=0:2
            A(e1+i) = A(e1+i) + Aloc(4+i);
        end
```

```
if e2isboundary && forcebound
    for i=0:2
        A(e2+i) = xi(e2+i);
    end
else
    for i=0:2
        A(e2+i) = A(e2+i)+ Aloc(7+i);
    end
end

if e3isboundary && forcebound
    for i=0:2
        A(e3+i) = xi(e3+i);
    end
else
    for i=0:2
        A(e3+i) = A(e3+i) + Aloc(10+i);
    end
end

% assemble global DA from local contributions
for i=0:2
    for j = 0:2
        iarray(arrayindex) = k+i;
        jarray(arrayindex) = k+j;
        valuearray(arrayindex) = DAloc(1+i,1+j);
        arrayindex = arrayindex + 1;
        iarray(arrayindex) = k+i;
        jarray(arrayindex) = e1+j;
        valuearray(arrayindex) = DAloc(1+i,4+j);
        arrayindex = arrayindex + 1;
        iarray(arrayindex) = k+i;
        jarray(arrayindex) = e2+j;
        valuearray(arrayindex) = DAloc(1+i,7+j);
        arrayindex = arrayindex + 1;
        iarray(arrayindex) = k+i;
        jarray(arrayindex) = e3+j;
        valuearray(arrayindex) = DAloc(1+i,10+j);
        arrayindex = arrayindex + 1;
    end
end
for i=0:2
    iarray(arrayindex) = e1+i;
    jarray(arrayindex) = k+i;
    valuearray(arrayindex) = DAloc(4+i,1+i);
    arrayindex = arrayindex + 1;
    iarray(arrayindex) = e1+i;
    jarray(arrayindex) = e1+i;
    valuearray(arrayindex) = DAloc(4+i,4+i);
    arrayindex = arrayindex + 1;
    iarray(arrayindex) = e1+i;
    jarray(arrayindex) = e2+i;
    valuearray(arrayindex) = DAloc(4+i,7+i);
    arrayindex = arrayindex + 1;
    iarray(arrayindex) = e1+i;
    jarray(arrayindex) = e3+i;
    valuearray(arrayindex) = DAloc(4+i,10+i);
    arrayindex = arrayindex + 1;
end

if e2isboundary && forcebound
    for i=0:2
        iarray(arrayindex) = e2+i;
        jarray(arrayindex) = e2+i;
        valuearray(arrayindex) = 1;
        arrayindex = arrayindex + 1;
    end
else
    for i=0:2
        iarray(arrayindex) = e2+i;
        jarray(arrayindex) = k+i;
```

```
                valuearray(arrayindex) = DAloc(7+i,1+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e2+i;
                jarray(arrayindex) = e1+i;
                valuearray(arrayindex) = DAloc(7+i,4+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e2+i;
                jarray(arrayindex) = e2+i;
                valuearray(arrayindex) = DAloc(7+i,7+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e2+i;
                jarray(arrayindex) = e3+i;
                valuearray(arrayindex) = DAloc(7+i,10+i);
                arrayindex = arrayindex + 1;
            end
        end

        if e3isboundary && forcebound
            for i=0:2
                 iarray(arrayindex) = e3+i;
                 jarray(arrayindex) = e3+i;
                 valuearray(arrayindex) = 1;
                 arrayindex = arrayindex + 1;
            end
        else
            for i=0:2
                iarray(arrayindex) = e3+i;
                jarray(arrayindex) = k+i;
                valuearray(arrayindex) = DAloc(10+i,1+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e3+i;
                jarray(arrayindex) = e1+i;
                valuearray(arrayindex) = DAloc(10+i,4+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e3+i;
                jarray(arrayindex) = e2+i;
                valuearray(arrayindex) = DAloc(10+i,7+i);
                arrayindex = arrayindex + 1;
                iarray(arrayindex) = e3+i;
                jarray(arrayindex) = e3+i;
                valuearray(arrayindex) = DAloc(10+i,10+i);
                arrayindex = arrayindex + 1;
            end
        end
    end
  end
end

DA = sparse(iarray,jarray,valuearray,L,L);
```

get_Aloc.m

```
function [Aloc,DAloc] = get_Aloc(kappa,h,k,e1,e2,e3,sk,x0,xi,t0,t1)
% Calculates contribution to A and to the Jacobian DA for a given cell K
%
%   kappa        time step width
%   h            cell width
%   x0           global coefficient vector at t=n
%   xi           newton-iterated global coefficient vector,
%                should eventually converge to global solution at t=n+1
%   k            index of the coefficient in the global coefficient vector
%                corresponding to the x-direction basis function on cell K,
%                k+1, k+2 then correspond to the y and z direction basis
%                functions on cell K.
%   e1, e2, e3   indices of the coefficients in the global coefficient
%                vector corresponding to the x-direction basis functions on
%                the edges of K. en+1, en+2 with n=1,2,3 then correspond to
%                the y and z directions basisfunction indices on these
%                edges. e1 denotes the diagonal edge.
%   sk           holds the sign of the thomas raviart fe which is
%                1   if the cell is a lower triangle and
%               -1  if the cell is a upper triangle
%
%   Aloc =      [ A_kx  ]
%               [ A_ky  ]
%               [ A_kz  ]
%               [ A_e1x ]
%               [ A_e1y ]
%               [ A_e1z ]
%               [ A_e2x ]
%               [ A_e2y ]
%               [ A_e2z ]
%               [ A_e3x ]
%               [ A_e3y ]
%               [ A_e3z ]
%
%   DAloc =     [ dA_ki/dmu_kj   dA_ki/dpi_e1j   dA_ki/dpi_e2j   dA_ki/dpi_e3j  ]
%               [ dA_e1i/dmu_kj  dA_e1i/dpi_e1j  dA_e1i/dpi_e2j  dA_e1i/dpi_e3j ]
%               [ dA_e2i/dmu_kj  dA_e2i/dpi_e1j  dA_e2i/dpi_e2j  dA_e2i/dpi_e3j ]
%               [ dA_e3i/dmu_kj  dA_e3i/dpi_e1j  dA_e3i/dpi_e2j  dA_e3i/dpi_e3j ]
%


% debugging flags
mterm1 = true;   %(d/dt(m),v)
mterm2 = true;   %(div(j),v,m*m)
mterm3 = true;   %(div(j)*m,m*v)
jterm1 = true;   %(div(q),m)
jterm2 = true;   %(j,q)
dAflag = true;   %calculate DF

% intialize local f and DF
Aloc  = zeros(12,1);
DAloc = zeros(12);

% precalculate some values
hsq2k    = h^2/(2*kappa);
skh      = sk*h;
skh2     = sk*h/2;
sksqrt2h = sk*sqrt(2)*h;
hsq3     = h^2/3;
hsq6     = h^2/6;

mpi  = 0.5 *   [ ( sqrt(2)*(xi(e1)   + x0(e1)  ) + xi(e2)   + x0(e2)   + xi(e3)   + x0(e3)   )   ; ...
                 ( sqrt(2)*(xi(e1+1) + x0(e1+1)) + xi(e2+1) + x0(e2+1) + xi(e3+1) + x0(e3+1) )   ; ...
                 ( sqrt(2)*(xi(e1+2) + x0(e1+2)) + xi(e2+2) + x0(e2+2) + xi(e3+2) + x0(e3+2) ) ] ;

mmu  = 0.5 *   [ (x0(k)   + xi(k)  )   ; ...
                 (x0(k+1) + xi(k+1))   ; ...
                 (x0(k+2) + xi(k+2)) ] ;
```

```
dtmu = hsq2k * [ (xi(k)   - x0(k)  )   ; ...
                 (xi(k+1) - x0(k+1))   ; ...
                 (xi(k+2) - x0(k+2)) ] ;


% assemble local A
if mterm1
    for i=1:3
        Aloc(i) = Aloc(i) + dtmu(i);
    end
end

if mterm2
    for i=1:3
        Aloc(i) = Aloc(i) - skh * mpi(i) * (mmu(1)^2 + mmu(2)^2 + mmu(3)^2);
    end
end

if mterm3
    for j=0:2
        i=j+1;
        ii=mod(j+1,3)+1;
        iii=mod(j+2,3)+1;
        Aloc(i) = Aloc(i) + skh * ( (mpi(i)*mmu(i) + mpi(ii)*mmu(ii) + mpi(iii)*mmu(iii)) * mmu(i) );
    end
end

if jterm1
    for i=0:2
        %rows corresponding to edge 1
        Aloc(4+i) = sksqrt2h * xi(k+i);
        %rows corresponding to edge 2
        Aloc(7+i) = skh * xi(k+i);
        %rows corresponding to edge 3
        Aloc(10+i) = skh * xi(k+i);
    end
end

if jterm2
    for i=0:2
        %rows corresponding to edge 1
        Aloc(4+i) = Aloc(4+i) + hsq3 * xi(e1+i);
        %rows corresponding to edge 2
        Aloc(7+i) = Aloc(7+i) + hsq3 * xi(e2+i) - hsq6 * xi(e3+i);
        %rows corresponding to edge 3
        Aloc(10+i) = Aloc(10+i) + hsq3 * xi(e3+i) - hsq6 * xi(e2+i);
    end
end


% assemble local DA
if dAflag

    % rows corresponding to basisfunctions on cells
    % df_ki/dmu_ki:
    DAloc(1,1) = hsq2k + skh2 * (mpi(2)*mmu(2) + mpi(3)*mmu(3)) ;
    DAloc(2,2) = hsq2k + skh2 * (mpi(1)*mmu(1) + mpi(3)*mmu(3)) ;
    DAloc(3,3) = hsq2k + skh2 * (mpi(1)*mmu(1) + mpi(2)*mmu(2)) ;

    % dA_ki/dmu_kj:
    DAloc(1,2) = skh2 * mpi(2)*mmu(1) - skh * mpi(1)*mmu(2);
    DAloc(1,3) = skh2 * mpi(3)*mmu(1) - skh * mpi(1)*mmu(3);
    DAloc(2,1) = skh2 * mpi(1)*mmu(2) - skh * mpi(2)*mmu(1);
    DAloc(2,3) = skh2 * mpi(3)*mmu(2) - skh * mpi(2)*mmu(3);
    DAloc(3,1) = skh2 * mpi(1)*mmu(3) - skh * mpi(3)*mmu(1);
    DAloc(3,2) = skh2 * mpi(2)*mmu(3) - skh * mpi(3)*mmu(2);

    % dA_kx/dpi_eax
    dA = -skh2 * (mmu(2)^2 + mmu(3)^2);
    DAloc(1,4) = sqrt(2) * dA;
    DAloc(1,7) = dA;
```

```
DAloc(1,10) = dA;

% dA_ky/dpi_eay
dA = -skh2 * (mmu(1)^2 + mmu(3)^2);
DAloc(2,5) = sqrt(2) * dA;
DAloc(2,8) = dA;
DAloc(2,11) = dA;

% dA_kz/dpi_eaz
dA = -skh2 * (mmu(1)^2 + mmu(2)^2);
DAloc(3,6) = sqrt(2) * dA;
DAloc(3,9) = dA;
DAloc(3,12) = dA;

% dA_kx/dpi_eay , dA_ky/dpi_eax
dA = skh2 * mmu(1)*mmu(2);%
DAloc(1,5) = sqrt(2) * dA;
DAloc(2,4) = sqrt(2) * dA;
DAloc(1,8) = dA;
DAloc(2,7) = dA;
DAloc(1,11) = dA;
DAloc(2,10) = dA;

% dA_kx/dpi_eaz , dA_kz/dpi_eax
dA = skh2 * mmu(1)*mmu(3);%
DAloc(1,6) = sqrt(2) * dA;
DAloc(3,4) = sqrt(2) * dA;
DAloc(1,9) = dA;
DAloc(3,7) = dA;
DAloc(1,12) = dA;
DAloc(3,10) = dA;

% dA_ky/dpi_eaz , dA_kz/dpi_eay
dA = skh2 * mmu(2)*mmu(3);%
DAloc(2,6) = sqrt(2) * dA;
DAloc(3,5) = sqrt(2) * dA;
DAloc(2,9) = dA;
DAloc(3,8) = dA;
DAloc(2,12) = dA;
DAloc(3,11) = dA;


% rows corresponding to basisfunctions on edges
% dA_e1i/dmu_ki
for i=0:2
    DAloc(4+i,1+i) = sksqrt2h;
end
% dA_e2i/dmu_ki
for i=0:2
    DAloc(7+i,1+i) = skh;
end
%dA_e3i/dmu_ki
for i=0:2
    DAloc(10+i,1+i) = skh;
end

% dA_e1i/dpi_e1i
for i=0:2
   DAloc(4+i,4+i) = hsq3;
end
% dA_e2i/dpi_e2i
for i=0:2
   DAloc(7+i,7+i) = hsq3;
end
% dA_e3i/dpi_e3i
for i=0:2
   DAloc(10+i,10+i) = hsq3;
end


% dA_e2i/dpi_e3i
```

```
    for i=0:2
        DAloc(7+i,10+i) = -hsq6;
    end
    % dA_e3i/dpi_e2i
    for i=0:2
        DAloc(10+i,7+i) = -hsq6;
    end

end
```

References

[1] *Iterative template library.* http://www.osl.iu.edu/research/itl/.

[2] X. S. Li and J. W. Demmel, *Superlu_dist: A scalable distributed-memory sparse direct solver for unsymmetric linear systems*, ACM Trans. Mathematical Software, 29 (2003), pp. 110–140.

[3] P. Sonneveld, *Cgs: A fast lanczos-type solver for nonsymmetric linear systems*, SIam J. Sci. Comput., 10 (1989), pp. 36–52.