Technical University of Vienna

Bachelor thesis

Numerical computation for the incompressible Navier Stokes equations and the coupling with the convection diffusion equation for the temperature

Author:
Philip
Lederer

Supervisor:
Prof. Dr. Joachim
Schöberl

2013

Abstract

This thesis discusses the Navier-Stokes equations, a non-linear partial differential equation describing the velocity of an incompressible fluid. Furthermore, those equations will be connected with the convection diffusion equation for the temperature to take a closer look on the phenomenon of natural convection.

The first section deals with the modelling of those partial differential equations including the Boussinesq approximation for the change of density which will occur by different temperatures in the fluid.

The second section will describe how to solve those equations using the computer. We will talk about mixed methods for the Navier-Stokes equations and the discontinuous galerkin method for the convection diffusion equation of the temperature. As those two equations are both time-dependent we will also discuss two different ways for the time discretization.

Section three and four contain practical examples. It is shown how to implement the discussed equations in NGSolve and will compare our results with the data given by the papers [5] and [8].

Acknowledgements

This thesis would not have been possible without the help and countless suggestions of Prof. Dr. Joachim Schöberl. Thanks for always keeping your doors open and spending time in helping me out of problems. Also i really want to thank him for giving me the chance to handle such a great topic, and that he always tried to focus on my interests.

Additionally I want to thank my colleagues and the whole team of the Institute for Analysis and Scientific Computing at the Technical University of Vienna for giving me a helping hand when it was needed.

Contents

List of Figures

1 Modelling

In the first section we focus on the modelling of the convection diffusion equation for the temperature, the Navier Stokes equations for the velocity of fluids and the Boussinesq approximation for density variations to describe the natural convection phenomenon. Therefor, we will need two theorems:

Theorem 1: (*Gauss Theorem*) For $V \subset \mathbb{R}^3$ compact with a piecewise smooth boundary ∂V, and F in $C^1(U, R^3)$ for an open subset $U \subset V$ we have:

$$\int_V \operatorname{div}(F)\, dx = \int_{\partial V} F \cdot n\, ds \tag{1}$$

Theorem 2: (*Transport theorem of Osborne Reynolds*) For each continuous and on Ω differentiable function $\Phi : \Omega \times [0, \infty] \to \mathbb{R}$ and a vector field $b : \mathbb{R}^3 \to \mathbb{R}^3$ we have for all $V(t) \subset \Omega$:

$$\frac{\partial}{\partial t} \int_{V(t)} \Phi(x,t)\, dx = \int_{V(t)} \left(\frac{\partial \Phi}{\partial t} + \operatorname{div}(\Phi b) \right)(x,t)\, dx \tag{2}$$

1.1 Convection diffusion equation for the temperature

In search for an equation for the evolution of the temperature $T(x,t)$ in an homogeneous bounded domain Ω with $x \in \Omega$, $t > 0$, first a control volume $V \subset \Omega$ is examined closer. Because of the conservation theorem of heat, that in fact describes the conservation of energy, we have:

$$\text{heat increase in } V = -\text{heatflux through } \partial V + \text{heat produced in } V.$$

The heat-energy itself is given by the integral over the temperature and a material dependent constant, the specific heat C. As C has the SI unit J/(kgK), and we have an integral over the volume V, we have to add the density ρ

$$\int_V C\rho T(x,t)\, dx.$$

So, the increase of the heat is given by the derivative by time

$$\text{heat increase in } V = \frac{d}{dt} \int_V C\rho T(x,t)\, dx = \int_V C\rho T_t(x,t)\, dx. \tag{3}$$

The heat produced in V should be given by a heat-source with density function $h(x,t)$.

$$\text{heat produced in } V = \int_V h(x,t)\, dx. \tag{4}$$

Finally, we take a closer look on the heat through ∂V. We define the function $q(x,t)$ as the heat flux, such that

$$\text{heat through } \partial V = \int_{\partial V} q \cdot n\, ds. \tag{5}$$

To describe the heat flux we use Fourier's law, which states that the heat flux is proportional to the negative gradient of the temperature, and include a term that describes the flow speed. We get the constitutative equation

$$q = -\kappa \nabla T + C\rho bT \tag{6}$$

with the thermal conductivity κ and a vector field b. Combining (3), (4), (5) and (6) we get

$$\int_V C\rho T_t(x,t)\, dx = \int_{\partial V} (\kappa \nabla T - C\rho bT) \cdot n\, ds + \int_V h(x,t)\, dx.$$

This equation can be simplified by using Gauss's theorem (1) on the surface integral yielding

$$\int_V C\rho T_t(x,t)\, dx = \int_V \operatorname{div}(\kappa \nabla T)\, dx - \int_V C\rho \operatorname{div}(bT)\, dx + \int_V h(x,t)\, dx.$$

As we assume enough smoothness of the integrated functions, and because the Gauss theorem is legit for all subsets $V \subset \Omega$ we may integrate over Ω and get the strong form:

$$C\rho T_t(x,t) = \operatorname{div}(\kappa \nabla T) - C\rho \operatorname{div}(bT) + h(x,t) \quad \text{in } \Omega$$

As we have a homogeneous domain, κ is constant and we can write that outside the divergence term. After dividing by $C\rho$ and pushing the divergence terms on the left side we get the partial differential equation for the temperature

$$\frac{\partial T}{\partial t} - \lambda \Delta T + \operatorname{div}(bT) = q \quad \text{in } \Omega \tag{7}$$

with $\lambda = \frac{\kappa}{C\rho}$ and $q = \frac{h}{C\rho}$. See [7] [9].

1.2 Navier Stokes equations

The Navier Stokes equations, named after Claude-Louis Navier and George Gabriel Stokes, is a non-linear partial differential equation that describes the motion and the velocity $u(x,t)$, of fluids with $x \in \Omega$ and $t > 0$. In our case, we want to focus on incompressible fluids and again use a control volume $V(t) \subset \Omega$ for the modelling.
From the point of view of a physicist, the first notion to be discussed is the conservation of mass

$$\int_{V(t)} \rho(x,t)\, dx = \text{const} \quad \forall t \in [0,\infty). \tag{8}$$

Using the transport theorem (2) with $b(x,t) = u(x,t)$, which is the velocity of the fluid, and $\Phi(x,t) = \rho(x,t)$, the derivative with respect to time on the left side becomes 0, and consequently

$$0 = \frac{\partial \rho}{\partial t} + \operatorname{div}(\rho u) \tag{9}$$

in $V \times [0,\infty)$. The transport theorem (2) is valid for all subsets $V \subset \Omega$, so (9) applies on $\Omega \times [0,\infty)$. Because we have an homogeneous fluid, ρ is constant, and the derivative again becomes 0. We get the continuity equation

$$\operatorname{div}(u) = 0. \tag{10}$$

This equation says, that no mass is produced in Ω as time elapses. Next we consider the conservation of momentum

$$m(t) = \int_V \rho(x,t)u(x,t)\ dx$$

and formulate Newtons second law

$$\frac{d}{dt}m(t) = F,$$

where F is the force acting on an object. This force can be divided into the force f which occurs in V

$$\int_V \rho(x,t)f(x,t)\ dx$$

and a force which occurs on the surface ∂V

$$\int_{\partial V} \sigma(x,t)\cdot n(x,t)\ ds,$$

where σ is the stress tensor. Thus brings us to

$$\frac{d}{dt}m(t) = \int_V \rho(x,t)f(x,t)\ dx + \int_{\partial V} \sigma(x,t)\cdot n(x,t)\ ds. \tag{11}$$

In the next step we use the transport theorem (2) on each component of the velocity on the left side of (11)

$$\frac{d}{dt}\int_V \rho u_i\ dx = \int_V \left(\frac{\partial}{\partial t}(\rho u_i) + \operatorname{div}(\rho u_i u)\right)(x,t)\ dx.$$

and the Gauss theorem (1) on the right side of (11)

$$\frac{d}{dt}m(t) = \int_V \rho(x,t)f(x,t) + \operatorname{div}(\sigma(x,t))\ dx.$$

Assuming enough smoothness of the functions, we integrate over Ω and because the Gauss theorem (1) and the transport theorem (2) are legit for all $V \subset \Omega$ we get the strong form:

$$\frac{\partial}{\partial t}(\rho u) + (\operatorname{div}(\rho u_i u))_{i=1:3} = \rho f + div(\sigma). \tag{12}$$

Again, because ρ is constant, we can simplify the divergence term of the velocity in (12) to

$$\operatorname{div}(\rho u_i u) = \rho\operatorname{div}(u_i u) = \rho\left(\nabla u_i \cdot u + u_i \underbrace{\operatorname{div}(u)}_{=0}\right) = \rho\nabla u_i \cdot u,$$

and we get the convection term

$$(\operatorname{div}(\rho u_i u))_{i=1:3} = \rho\begin{pmatrix} u\cdot\nabla u_1 \\ u\cdot\nabla u_2 \\ u\cdot\nabla u_3 \end{pmatrix} = \rho(u\cdot\nabla)u. \tag{13}$$

As the fluid is viscose, the stress tensor σ can be further refined, to calculate the $\operatorname{div}(\sigma)$ term in the strong formulation (12). Thanks to physics we know that

$$\sigma(x,t) = -p(x,t)I + \tau(\nabla u(x,t)),$$

where $I := (\delta_{ij})_{i,j=1:3}$ is the unit tensor, and p is the pressure. For a Newtonian fluid, τ has to achieve the following requirements:

- τ is just linear dependant on the gradient of the velocity
- τ is symmetric
- τ is invariant in rotation

By using that information, physics tells us that

$$\tau(\nabla u) = \lambda(\mathrm{div}(u))I + 2\nu D(u)$$

where

$$D(u) = \frac{1}{2}(\nabla u + (\nabla u)^T) = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right)_{i=1:3}$$

is the deformation tensor, and λ, ν are the dynamic viscosities. So we get

$$\begin{aligned}
\mathrm{div}(\sigma) &= \mathrm{div}\left(-pI + \lambda(\underbrace{\mathrm{div}(u)}_{=0})I + 2\nu D(u)\right) && (14)\\
&= -\mathrm{div}(pI) + \nu\mathrm{div}\left((\nabla u + (\nabla u)^T)\right) && (15)\\
&= -\mathrm{div}(pI) + \nu\mathrm{div}\left(\nabla u\right) + \mathrm{div}\left((\nabla u)^T\right) && (16)\\
&= -\nabla p + \nu\Delta u. && (17)
\end{aligned}$$

By combining the continuity term (10), the strong formulation (12), the convection term (13) and the divergence of the stress tensor (17) we finally get the Navier Stokes equations

$$\begin{aligned}
\rho\frac{\partial u}{\partial t} - \nu\Delta u + \rho(u\cdot\nabla)u + \nabla p &= \rho f\\
\mathrm{div}(u) &= 0
\end{aligned} \qquad (18)$$

See [2] [3].

1.3 Natural convection

The natural convection is a problem arising from the coupling of the Navier Stokes equations discribing the motion of the fluid, and the convection diffusion equation for the temperature. In fact the forces which induce natural convection are produced due to variable gravity forces that occur because of the variation of the density in the fluid affected by the non-uniformity of the temperature.
More precisely, it is intended to model heat that is applied on the bottom, and cooling that is applied on the top, what will produce a lifting force that will act against the gravity force g. To model that lifting force we use the Boussinesq approximation, which says that density differences are so small that they can be neglected except in terms where they appear in a multiplication with g. Therefore we will use

$$g' = g\frac{\rho_1 - \rho_2}{\rho}$$

instead of g. ρ could be approximated by either ρ_1 or ρ_2 because the difference $\Delta\rho = \rho_1 - \rho_2$ is really small.

In our example we will calculate the difference of the densities by using a linear formulation including the temperature

$$g' = g\rho_0(1 - \beta(T - T_0))$$

where ρ_0 is a reference density and T_0 is a reference temperature. So with ρ_0 and g' as a force in (39) we get:

$$\rho_0 \frac{\partial u}{\partial t} - \nu\Delta u + \rho_0(u \cdot \nabla)u + \nabla p = g\rho_0(1 - \beta(T - T_0))$$

and after dividing by ρ_0

$$\frac{\partial u}{\partial t} - \frac{\nu}{\rho_0}\Delta u + (u \cdot \nabla)u + \frac{\nabla p}{\rho_0} = g(1 - \beta(T - T_0)),$$

with g as a vector $g = \begin{pmatrix} 0 \\ -9.81 \end{pmatrix}$. For the temperature T we will use (7) with the velocity u for the vector field b

$$\frac{\partial T}{\partial t} - \lambda\Delta T + \operatorname{div}(uT) = q. \tag{19}$$

As can be seen, the two partial differential equations are coupled. T is used in the Navier Stokes equations for gravity force variations, and the velocity u as a vector field in the convection diffusion equation for the temperature. See [5].

2 Discretization methods

In this section we want to discuss how to get a proper numerical solution for our equations. For that, we introduce different methods and deal with the problem of a uniquely and stable solution.

2.1 Mixed methods

The main application for mixed methods are finite element methods where two spaces are used to approximate two different variables. Quite often the second variable is introduced in the formulation because of the physical interest and is usually related to some derivatives of the original variable like in the elasticity equation. In other cases there are two natural independent variables as in the Navier Stokes equations, where we have the velocity and the pressure. In such cases, mixed methods seem to be the most natural solution strategy.

2.1.1 Abstract theory

A mixed variational formulations implies two Hilbert spaces V and Q, bilinear-foms

$$a(u, v) : V \times V \to \mathbb{R}$$
$$b(u, q) : V \times Q \to \mathbb{R}$$

and continuous linear-forms

$$f(v) : V \to \mathbb{R}$$
$$q(q) : Q \to \mathbb{R}$$

The problem is to find $u \in V$ and $p \in Q$ such that

$$\begin{aligned} a(u,v) + b(v,p) &= f(v) \quad \forall v \in V \\ b(u,q) &= g(q) \quad \forall q \in Q \end{aligned}$$

Contrary to the ordinary case, where we have a system of equations, we now want to combine them and look at the mixed method as variational problem on the product space $V \times Q$. This can be done by simply adding the two lines. So now we want to find a tuple $(u,p) \in V \times Q$ such that

$$a(u,v) + b(u,q) + b(v,p) = f(v) + g(q) \quad \forall (v,q) \in V \times Q$$

Let us define the big bilinear-form $B(\cdot,\cdot) : (V \times Q) \times (V \times Q) \to \mathbb{R}$ as

$$B((u,p),(v,q)) = a(u,v) + b(u,q) + b(v,p),$$

so we can write the mixed method as single variational problem:

$$\text{Find } (u,p) \in V \times Q : \quad B((u,p),(v,q)) = f(v) + g(q) \quad \forall (v,q) \in V \times Q$$

See [10].

2.1.2 LBB-condition and the Brezzi Theorem

The fundamental question arises, if there is a stable and uniquely solution for the mixed problem. This is given by the work of the Russian mathematician Olga Alexandrowna Ladyschenskaja, the Czech-American mathematician Ivo Babuška and the Italian mathematician Franco Brezzi. They showed, that under the some well defined conditions, a saddle point problem, like that one we have, provides a uniquely stable solution. To see this, some linear operators provided by the Riesz representation theorem are introduced:

$$\begin{aligned} A: &\quad V \to V, u \to Au \quad \text{so that} \quad (Au,v)_V = a(u,v) \quad \forall v \in V \\ B: &\quad V \to Q, u \to Bu \quad \text{so that} \quad (Bu,q)_Q = b(u,q) \quad \forall q \in Q \\ B^*: &\quad Q \to V, p \to B^*p \quad \text{so that} \quad (B^*p,v)_V = b(v,p) \quad \forall v \in V \end{aligned}$$

By this, the mixed variational problem can be written as operator equation

$$\begin{aligned} Au + B^*p &= J_V f \\ Bu &= J_Q g \end{aligned}$$

using the Riesz-Isomorphismus $J_V : V* \to V$ and $J_Q : Q* \to Q$, or also

$$\begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix} \begin{pmatrix} u \\ p \end{pmatrix} = \begin{pmatrix} J_V f \\ J_Q g \end{pmatrix}.$$

Such linear equation systems including a block matrix of the form

$$\begin{pmatrix} A & B^* \\ B & 0 \end{pmatrix}$$

are called saddle-point problems. The Brezzi Theorem, including the LBB-condition, states the precise requirements to guarantee a uniquely stable solution.

Theorem 3: (*Brezzi's theorem*) Assume that $a(.,.)$ and $b(.,.)$ are continuous bilinear-forms

$$\begin{aligned} a(u,v) &\leq \alpha_2 \|u\|_V \|v\|_V \quad \forall u, v \in V \\ b(u,q) &\leq \beta_2 \|u\|_V \|q\|_Q \quad \forall u \in V, \forall q \in Q. \end{aligned}$$

Assume there holds coercivity of $a(.,.)$ on the kernel, i.e.,

$$a(u,u) \geq \alpha_1 \|u\|_V^2 \quad \forall u \in V_0$$

with $V_0 := \{v : Bv = 0\}$, and there holds the LBB-condition

$$\sup_{u \in V} \frac{b(u,q)}{\|u\|_V} \geq \beta_1 \|q\|_Q \quad \forall q \in Q. \tag{20}$$

Then the mixed problem is uniquely solvable, and the solution fullfills the stability estimate

$$\|u\|_V + \|p\|_Q \leq c\{\|f\|_{V^*} + \|g\|_{Q^*}\},$$

with the constant c depending on $\alpha_1, \alpha_2, \beta_1, \beta_2$. See [10].

2.1.3 Stokes equation

Next we choose a simpler model problem, the Stokes equations, which are in fact the same as the Navier Stokes equations, but used for incompressible fluids with a high viscosity. In this case the divergence term can be neglected, and so it is easier to analyse them. Assume $\Omega \subset \mathbb{R}^2$ with a piecewise smooth boundary, then the Stokes equations can be written as:

$$\begin{aligned} -\nu \Delta u + \nabla p &= f \quad \text{in } \Omega \\ \operatorname{div}(u) &= 0 \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \partial\Omega \end{aligned} \tag{21}$$

Again $u : \Omega \to \mathbb{R}$ is the velocity and $p : \Omega \to \mathbb{R}$ is the pressure. When there exist functions $u \in [C^2(\Omega) \cap C^0(\overline{\Omega})]^2$ and $p \in C^1(\Omega)$, then u and p are called classical solutions. Since we have Dirichlet boundary conditions, the pressure is thereby defined up to an additive constant. Usually, the standardization

$$\int_\Omega p \, dx = 0 \tag{22}$$

is used.
To get the weak formulation, we multiply the first line with test-functions $v \in [H_0^1(\Omega)]^2$,

the second line with test-functions $q \in L_0^2(\Omega) := \{q \in L^2(\Omega) : \int_\Omega q\, dx = 0\}$

$$\begin{aligned} \int_\Omega -\nu \overbrace{\operatorname{div}(\nabla u)}^{=\Delta u} v + \int_\Omega \nabla p v &= \int_\Omega f v \\ \int_\Omega \operatorname{div}(u) q &= 0 \end{aligned}$$

integrate by parts

$$\begin{aligned} \int_\Omega \nu \nabla u \nabla v + \overbrace{\int_{\partial\Omega} v \nabla u n\, ds}^{=0} - \int_\Omega \operatorname{div}(v) p + \overbrace{\int_{\partial\Omega} p(v \cdot n)\, ds}^{=0} &= \int_\Omega f v \\ \int_\Omega \operatorname{div}(u) q &= 0 \end{aligned}$$

and use $-p$ as value for the pressure (this is not affecting the solution) to get

$$\begin{aligned} \int_\Omega \nu \nabla u \nabla v + \int_\Omega \operatorname{div}(v) p &= \int_\Omega f v \\ \int_\Omega \operatorname{div}(u) q &= 0. \end{aligned} \tag{23}$$

Together with

$$\begin{aligned} a(u, v) &:= \int_\Omega \nu \nabla u \nabla v\, dx, \\ b(u, q) &:= \int_\Omega \operatorname{div}(u) q\, dx, \\ (f, v) &:= \int_\Omega f v\, dx, \end{aligned}$$

we get the the saddle point problem: Find $(u, p) \in V \times Q$ such that

$$\begin{aligned} a(u, v) + b(v, p) &= (f, v) \quad \forall v \in V \\ b(u, q) &= 0 \quad \forall q \in Q \end{aligned}$$

See [10].

2.1.4 Taylor-Hood element

In the last sections we have provided the theoretical results needed for the analysis of a Galerkin discretization for the stokes equation. The difficulty lies in finding spaces $V_h \subset V = [H_0^1(\Omega)]^2$ and $Q_h \subset Q = L_0^2(\Omega)$, so that the LBB-condition (20)

$$\sup_{u \in V_h} \frac{\int_\Omega \operatorname{div}(u) q}{\|u\|_{V_h}} \geq \beta_1 \|q\|_{Q_h} \quad \forall q \in Q_h.$$

is still valid. A quite famous element is the Taylor-Hood element (Figure 1) with the chosen spaces

$$\begin{aligned} V_h &:= \{v_h \in [C(\overline{\Omega})]^2 \cap [H_0^1(\Omega)]^2 : v_{h|T} \in P^k \text{ for } T \in \mathcal{T}_h\} \\ Q_h &:= \{q_h \in C(\Omega) \cap L_0^2(\Omega) : q_{h|T} \in P^{k-1} \text{ for } T \in \mathcal{T}_h\}. \end{aligned}$$

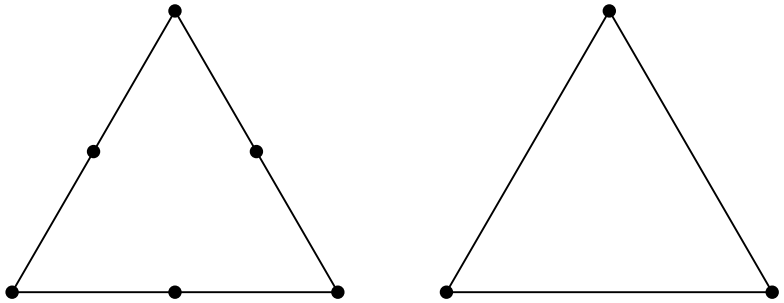

Figure 1: P^2 and P^1 Taylor-Hood elements

With the right triangulation $\mathcal{T}_h$, it can be shown, that for $k > 1$ the LBB-condition (20) is valid. For a proof we want to refer to [6, Chp. VI]. The other assumptions for Brezzi's theorem (2.1.2) can be shown with the common methods. See [1].

2.2 Discontinuous Galerkin Method

Next we want to introduce a finite element method used for numerical calculations of convection diffusion equations like (7) where the convection term is dominant. The difference to common methods is, that we choose finite element spaces that are piecewise polynomials and do not provide continuity across element boundaries

$$V_h^{DG} := \{v \in L^2 : v|_T \in P^k\} \tag{24}$$

2.2.1 The convection equation

Assume $\Omega \subset \mathbb{R}^2$ with a piecewise smooth boundary and a vectorfield $b : \mathbb{R}^2 \to \mathbb{R}^2$ with $\operatorname{div}(b) = 0$, then the convection equation is: Find $u : \mathbb{R} \to \mathbb{R}$ with

$$\begin{aligned} \operatorname{div}(bu) &= f \quad \text{in } \Omega \\ u &= u_D \quad \text{on } \partial\Omega. \end{aligned}$$

To get the weak formulation we first integrate on Ω and multiply with test functions $v \in L^2, v|_T \in H^1(T)$

$$\int_\Omega \operatorname{div}(bu)v = \int_\Omega (b \cdot \nabla u + u \overbrace{\operatorname{div}(b)}^{=0})v = \int_\Omega (b \cdot \nabla u)v = \int_\Omega fv.$$

Next we want to integrate by parts. To do that, we have to assume that the test functions are piece-wise smooth:

$$\int_\Omega (b \cdot \nabla u)v = \sum_T \left(- \int_T u(b \cdot \nabla v) + \int_{\partial T} \overbrace{u(b \cdot vn)}^{=uvb_n} \right) = \int_\Omega fv \quad \forall v \in L^2, v|_T \in H^1(T).$$

The question is, which value we use for u on the boundary ∂T. For that, we assume that $u \in V_h^{DG}$, and choose the upwind value

$$u^{up} := \begin{cases} u|_T & \text{for} \quad b_n > 0 \text{ (outflow boundary)} \\ u|_{T'} & \text{for} \quad b_n \leq 0 \text{ (inflow boundary) and } T' \text{neighbour triangle} \end{cases} \tag{25}$$

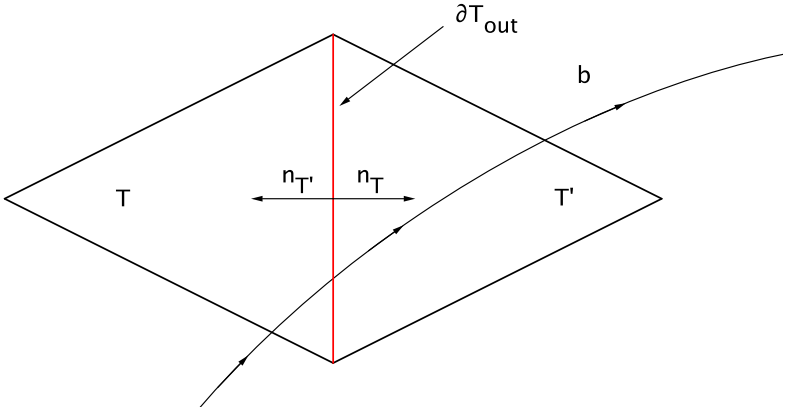

Figure 2: Element T and T'

If an edge $E \subset \partial T$ is on the domain inflow boundary $\Gamma_{in} \subset \partial\Omega$, the Dirichlet boundary value u_D is taken as upwind value (25). So now the weak formulation can be written as

$$\underbrace{\sum_T \left(-\int_T u(b \cdot \nabla v) + \int_{\partial T \setminus \Gamma_{in}} b_n u^{up} v \right)}_{=:A^{DG}_{conv}(u,v)} = \underbrace{\int_\Omega fv - \int_{\Gamma_{in}} b_n u_D v}_{=:f^{DG}(v)} \quad \forall v \in L^2, v|_T \in H^1(T).$$

and the DG finite element method is now: Find $u_h \in V_h^{DG}$ so that

$$A^{DG}_{conv}(u,v) = f^{DG}(v_h) \quad \forall v_h \in V_h^{DG}.$$

Next we want to reformulate A^{DG} by combining the integral of the element-outflow boundary of T with the neighbour boundary of T'. Consider that

$$\int_{\partial T'_{in}} b_{n_{T'}} u^{up} v_{T'} = -\int_{\partial T_{out}} b_{n_T} u|_T v_{T'}$$

on the shared boundary because $b_{n_{T'}} = -b_{n_T}$ (see figure 2). By doing that, we can just sum over all ∂T_{out} with the special case $v_{T'} = 0$ for $\partial T_{out} \subset \partial\Omega$ and get

$$A^{DG}_{conv}(u,v) = \sum_T \left(-\int_T u(b \cdot \nabla v) + \int_{\partial T_{out}} b_n u^{up} \underbrace{(v|_T - v|_{T'})}_{=:[v]} \right). \tag{26}$$

If we integrate by parts back on each element we see that

$$A^{DG}_{conv}(u,v) = \sum_T \left(\int_T (b \cdot \nabla u)v - \int_{\partial T} b_n uv + \int_{\partial T \setminus \Gamma_{in}} b_n u^{up} v \right) \tag{27}$$

$$= \sum_T \left(\int_T (b \cdot \nabla u)v + \int_{\partial T_{in}} b_n \underbrace{(u|_{T'} - u|_T)}_{-[u]} v \right). \tag{28}$$

By averaging the two formulations (26) and (28) we get the final form

$$A^{DG}_{conv}(u,v) = \frac{1}{2}\sum_T \left(\int_T \{(b \cdot \nabla u)v - u(b \cdot \nabla v)\} \right) + \frac{1}{2}\sum_{E \in \Omega^\circ} \left(\int_E |b_n|[u][v] \right) + \frac{1}{2}\int_{\partial\Omega} |b_n| uv \tag{29}$$

An example should illustrate our findings: Let E be an internal edge between T_1 (seen as the upwind triangle) and T_2 (seen as the downwind triangle). Then, T1 contributes from (26) the term on its outflow-edge

$$\frac{1}{2}b_{n_1}u_1(v_1 - v_2),$$

and T_2 contributes from (28) on its inflow-edge

$$\frac{1}{2}b_{n_2}(u_1 - u_2)v_2.$$

By adding those two terms and the fact that $0 < b_{n_1} = -b_{n_2}$, we get the term that occurs in (29)

$$\frac{1}{2}b_{n_1}(u_1(v_1 - v_2) - (u_1 - u_2)v_2) = \frac{1}{2}|b_{n_1}|[u][v].$$

If E is a boundary edge, then either (26) or (28) contributes the term $\frac{1}{2}|b_{n_1}|uv$. With this construction the method is exact for the correct $u \in [C^2(\Omega) \cap C^0(\overline{\Omega})]^2$, and stable in V_h^{DG}, therefore we get convergence. See [10].

2.2.2 The diffusion equation

Assume $\Omega \subset \mathbb{R}^2$ with a piecewise smooth boundary. Then the diffusion equation is: Find $u : \mathbb{R} \to \mathbb{R}$ with

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ u &= u_D \quad \text{on } \partial\Omega. \end{aligned}$$

Again we first integrate on Ω, multiply with test functions $v \in L^2, v|_T \in H^1(T)$

$$\int_\Omega -\Delta u v = \int_\Omega f v,$$

and integrate by part ($u \in V_h^{DG}$)

$$\sum_T \left(\int_T \nabla u \nabla v - \int_{\partial T} \underbrace{(\nabla u \cdot n)}_{\frac{\partial u}{\partial n}} v \right) = \int_\Omega f v \quad \forall v \in L^2, v|_T \in H^1(T).$$

As for the convection equation we use the definitions:

$$\begin{aligned} \text{Jump:} \quad & [u] := u_1 - u_2 \\ \text{Mean value:} \quad & \langle \tfrac{\partial u}{\partial n} \rangle := \frac{1}{2}\left(\frac{\partial u_1}{\partial n_1} + \frac{\partial u_2}{\partial n_2} \right) \cdot \end{aligned}$$

Next we do some transformations, stabilize and symmetrize to get:

$$\underbrace{\sum_T \left(\int_T \nabla u \nabla v \right) - \sum_E \left(\int_E \langle \frac{\partial u}{\partial n} \rangle [v] \right) - \sum_E \left(\int_E \langle \frac{\partial v}{\partial n} \rangle [u] \right) + \sum_E \frac{\alpha}{h} \int_E [u][v]}_{=: A_{diff}^{DG}(u,v)} = \int_\Omega f v \tag{30}$$

with a factor α. For the correct solution u, the jump $[u] = 0$, so when we integrate back by parts we get the diffusion equation again. Conventionally, A_{diff}^{DG} is elliptic on V_h^{DG}, provided that $\alpha = \mathcal{O}(1)$ is large enough. Again we get consistence and stability on V_h^{DG}, so convergence for our discretization. See [10].

2.3 Time discretization

Finally we want to find a way to get a discretization of time independent problems as we discussed in section (1). By using a differential operator $L(u)$ we can write the problem in the short form:

$$\begin{aligned} \frac{\partial u}{\partial t} + L(u) &= f(t) \quad \text{in } \Omega \times [0,T] \\ u &= u_D \quad \text{on } \partial\Omega \times [0,T] \\ u &= u_0 \quad \text{in } \Omega \times 0 \end{aligned}$$

For our methods we first want to solve the problem in Ω, so we look for a discretization of space, and then use a method for the time. To do that, we integrate over Ω and multiply with proper test functions v,

$$\int_\Omega \frac{\partial u}{\partial t} v + \int_\Omega L(u)v = \int_\Omega fv \quad \forall v \in V.$$

Together with the bilinear form a

$$a(u,v) = \int_\Omega L(u)v$$

and a linear form F

$$F(v) = \int_\Omega fv,$$

we get the formulation

$$\begin{aligned} \left(\frac{\partial u}{\partial t}, v\right)_{L^2} + a(u,v) = F(v) \quad \forall v \in V, \forall t \in [0,T] \\ u(0) = u_0. \end{aligned}$$

With a discretization $u_h : [0,T] \to V_h$, where $ndof$ is the number of degrees of freedom:

$$u_h(x,t) := \sum_{i=1}^{ndof} u_i(t)\phi_i(x), \tag{31}$$

we can write

$$\left(\frac{\partial}{\partial t} u_h, \phi_j\right)_{L^2} + a(u_h, \phi_j) = F(\phi_j) \quad \forall j = 1 \dots ndof, \forall t \in [0,T].$$

With $M_{ij} := \int_T \phi_i \phi_j$, the so called mass matrix, we can write our problem as:

$$\begin{aligned} M\underline{u}'(t) + A\underline{u}(t) &= f \quad \text{in } \Omega \times [0,T] \\ u &= u_0 \quad \text{in } \Omega \times 0 \end{aligned} \tag{32}$$

with the coefficient vector $\underline{u} = (u_i)_{i=1:ndof}$. For the ease of notation, we will write u_h for $\underline{u}$ from now on.

2.3.1 Implicit Euler method

One of the simplest ways is to use one of the Euler methods. Lets define a equidistant partitioning of the time interval

$$0 = t_0 < t_1 < \cdots < t_{m-1} < t_m = T$$

with $\tau := t_j - t_{j-1}$, and integrate over one segment

$$\int_{t_j}^{t_{j+1}} Mu_h'(s) + Au_h(s)\ ds = \int_{t_j}^{t_{j+1}} f(s)\ ds.$$

or

$$M\left(u_h(t_{j+1}) - u_h(t_j)\right) + \int_{t_j}^{t_{j+1}} Au_h(s)\ ds = \int_{t_j}^{t_{j+1}} f(s)\ ds.$$

Now we can choose between different numerical integration rules to replace the integrals. We use the right-side rule and with the notation $u_h(t_j) := u_j$ we get

$$M(u_{j+1} - u_j) + \tau A u_{j+1} = \tau f_{j+1}.$$

or

$$(M + \tau A)u_{j+1} = Mu_j + \tau f_{j+1}.$$

In case of a right-side rule, a linear system is solvable in any case. So we get

$$u_{j+1} = (M + \tau A)^{-1}(Mu_j + \tau f_{j+1}), \tag{33}$$

which is called the implicit Euler method. See [4].

2.3.2 Diagonal implicit Runge-Kutta method

A famous class of one-step methods for time discretization are the so called Runge-Kutta methods. We start with our equation (32), transform it into the integral form

$$\begin{aligned} Mu_h'(t) + Au_h(t) &= f(t) \\ Mu_h'(t) &= f(t) - Au_h(t) \\ u_h'(t) &= \underbrace{M^{-1}(f(t) - Au_h(t))}_{:=F(t,u_h(t))}, \end{aligned}$$

and integrate over one interval of the partitioning

$$\int_{t_j}^{t_{j+1}} u_h'(s)\ ds = \int_{t_j}^{t_{j+1}} F(s, u_h(s))\ ds$$

so that

$$u_h(t_{j+1}) = u_h(t_j) + \int_{t_j}^{t_{j+1}} F(s, u_h(s))\ ds. \tag{34}$$

Next we can choose between different quadrature rules for the right integral, including normed integration points $c_j \in [0, 1]$ and normed weights b_j for $j = 1, \ldots, s$. With that we get

$$u_h(t_{j+1}) = u_h(t_j) + \tau \sum_{l=1}^{s} b_l F(t_j + c_l\tau, u_h(t_j + c_l\tau)). \tag{35}$$

c	A
	b^T

Table 1: Scheme of Butcher tableau

The problem is, that we do not know the values of $u_h(t_j + c_l\tau)$, so again we use numeric integration

$$u_h(t_j + c_i\tau) = u_h(t_j) + \int_{t_j}^{t_j+c_i\tau} F(s, u_h(s))\, ds \approx u_h(t_j) + \tau \sum_{l=1}^{s} a_{il} F(t_j + c_l\tau, u_h(t_j + c_l\tau)).$$

Thus, a_{il} are the normed weights for

$$\int_0^{c_i} F(x)\, dx \approx \sum_{l=1}^{s} a_{il} F(c_l).$$

With $u_h(t_j) := u_j$ we finally get:

$$\begin{aligned} u_h(t_j + c_i\tau) := u_{j,i} &= u_j + \tau \sum_{l=1}^{s} a_{il} \underbrace{F(t_j + c_l\tau, u_{j,l})}_{=k_l} \quad i = 1, \ldots, s \\ u_{j+1} &= u_j + \tau \sum_{l=1}^{s} b_l F(t_j + c_l\tau, u_{i,l}) = u_j + \tau \sum_{l=1}^{s} b_i k_l \\ k_i &:= F(t_j + c_i\tau, u_j + \tau \sum_{l=1}^{s} a_{il} k_l) \end{aligned} \cdot$$

We want to focus on autonomous differential equations, because every non autonomous can be rewritten into an autonomous one, so $c_i = \sum_{l=1}^{s} a_{il}$. A good way to illustrate a Runge-Kutta method is to use a Butcher-tableau. There the coefficients a_{il}, b_j and c_j are displayed with a matrix $A \in \mathbb{R}^{s\times s}$ and two vectors $b \in \mathbb{R}^s$ and $c \in \mathbb{R}^s$. When we want to use an explicit method, then the matrix A has to be a strictly lower triangular matrix, so we have $\frac{s(s-1)}{2}$ coefficients for the a_{il} and s coefficients for the b_j. When we want to use an implicit method, we have to solve a linear system in every step. As we can see now, the implicit Euler method is also a Runge-Kutta method with $s = 1$ and the coefficients $c_1 = 1$, $b_1 = 1$ and $a_{11} = 1$.

The diagonal implizit Runge-Kutta methods, abbreviated by DIRK, have a lower diagonal matrix A with $a_{ii} = \alpha$ for $i = 1, \ldots, s$. The constant α is chosen for stability reasons. We want to have two integration points, so $s = 2$, and to get a L-stable method one row of A has to be b^T. Together with $c_i = \sum_{l=1}^{s} a_{il}$ we get the butcher tableau seen in table (2). The method should have the order 2, so we have to consider that

$$\sum_{i=1}^{s} b_i c_i = \frac{1}{2}$$

so

$$(1 - \alpha)\alpha + \alpha = \frac{1}{2}.$$

α	α	0
1	$1-\alpha$	α
	$1-\alpha$	α

Table 2: Butcher tableau for 2 step, L-Stable DIRK

This equation has the solutions $\alpha_1 = 1 - \sqrt{\frac{1}{2}}$ and $\alpha_2 = 1 + \sqrt{\frac{1}{2}}$. As $c_1 = \alpha$ is our first integration point, we choose the solution $\alpha = \alpha_1$, because otherwise we would jump out of the time interval τ. For the solution of the next step, no we have to compute the k_i

$$k_1 = F(t_j + \alpha\tau, u_j + \tau\alpha k_1) = M^{-1}(f_{t+\alpha\tau} - A(u_j + \tau\alpha k_1))$$
$$k_2 = F(t_j + \tau, u_j + \tau(1-\alpha)k_1 + \tau\alpha k_2) = M^{-1}(f_{t+\tau} - A(u_j + \tau(1-\alpha)k_1 + \tau\alpha k_2))$$

so

$$\begin{aligned} k_1 &= (M + A\tau\alpha)^{-1}(f_{t+\alpha\tau} - Au_j) \\ k_2 &= (M + A\tau\alpha)^{-1}(f_{t+\tau} - A(u_j + \tau(1-\alpha)k_1)) \end{aligned} \tag{36}$$

to get

$$u_{j+1} = u_j + \tau b_1 k_1 + \tau b_2 k_2$$

See [4].

3 Numerical computation using NGSOLVE

In the next step, the discussed methods shall be put into numerical practise. The programm NGSOLVE, which is a general purpose Finite Element Library on top of NETGEN, is complemented and used in all subsequent calculations.

3.1 PDE-File

Partial differential equations can be loaded with the usage of PDE files (*.pde), where all the information that describes the problem is saved. The first two lines indicate the used Ω (in our case $\mathbb{R}^2$), and will tell NGSOLVE which mesh, generated by NETGEN, is used. So we need a *.in2d file, representing the geometry, and a *.vol file created by NETGEN.

```
geometry = example.in2d
mesh = example.vol
```

Next, we have to define coefficients that are used in the partial differential equation, like the viscosity or the density. We want to set $\mu := -1$.

```
define coefficient mu
-1,
```

For the boundary conditions we also have to define coefficients. Therefore we have to

differ between using Dirichlet-, Neumann- or Robin-conditions. Although all three can be computed with NGSOLVE, we will just discuss Robin-conditions

$$\frac{\partial u}{\partial n} - \alpha(u - u_D) = g$$

By variation of α we can now decide if we either want to have Dirichlet- or Neumann-conditions. If $\alpha >> 1$, then u will be pushed down to the value u_D, and with $\alpha = 0$ we will get Neumann conditions. To see this, we make an example for the diffusion equation $-\Delta u = f$. We integrate by part over Ω to get the weak formulation, and use the boundary condition

$$\begin{aligned}\int_\Omega -\Delta uv &= \int_\Omega \nabla u \nabla v + \int_{\partial\Omega} \frac{\partial u}{\partial n} v = \int_\Omega fv \\ &= \int_\Omega \nabla u \nabla v + \int_{\partial\Omega} \alpha(u - u_D)v + gv = \int_\Omega fv \\ &= \int_\Omega \nabla u \nabla v + \int_{\partial\Omega} \alpha uv = \int_\Omega fv + \int_{\partial\Omega} \alpha u_D v - \int_{\partial\Omega} gv. \end{aligned} \tag{37}$$

We now just have to define two coefficients ubound, and penalty. Penalty will indicate where we want to have which boundary condition by setting α, and ubound will describe the values for either g or u_D. Lets say that $\partial\Omega = \Gamma_1 \cup \Gamma_2$, and we want to have

$$\begin{aligned} u &= 300 \quad \text{on } \Gamma_1 \\ \frac{\partial u}{\partial n} &= 0 \quad \text{on } \Gamma_2. \end{aligned}$$

As α also acts on u_D on the right side, we have to include this in the coefficients, so we get

```
define coefficient penalty
1e8, 0,
define coefficient ubound
(1e8*300), 0,
```

For the right side of the partial differential equation we set a coefficient that describes the function f, for example $f := x^2 + y^2$.

```
define coefficient rside
(x*x+y*y);
```

Next we will define a finite element space and a grid function that will illustrate our solution. For the finit element space different options can be taken. We want to use an H^1 with basic functions $\phi \in P^2$. Note that when the grid function is defined, the space on which the grid function acts, has to be included.

```
define fespace v -type=h1ho -order=2

define gridfunction u -fespace=v
```

Finally we have to define our bilinear forms and linear forms. They are both defined

by the integrators that stand below the definitions in the PDE-file. NGSOLVE already provides different integrators like a laplace integrator or a source integrator. By using Robin penalties we also have to add some other integrals, as seen in (37). For the left side an integrator called "robin" is provided, and for the right side we will use the integrator called "neumann" for the boundary integral. Now we need our coefficients we defined before, so that the integrators know where and with which coefficients they have to calculate.

```
define bilinearform a -fespace=v
laplace mu
robin penalty

define linearform f -fespace=v
source rside
neumann ubound
```

For solving the equation we have to define a preconditioner, the bilinearform, and a numerical procedure that is provided by NGSOLVE. We want to solve a bounday value problem, so we use the numerical procudeure "bvp" and set how many steps for the approximation of the solution should be performed:

```
define preconditioner c -type=direct -bilinearform=a

numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=u
-preconditioner=c -maxsteps=100
```

At the bottom of the file, some visualizations procedures will be called to visualize the solutions in NGSOLVE, but those wont be discussed in this thesis. Now the file is complete and can be loaded in NGSOLVE to solve the problem.

3.2 Creating new procedures and integrators

As not every integrator ore procedure is provided by NGSOLVE, there is a programming interface, that can be used to integrate new code. To do so, a library can be loaded with any C++ compiler, so all the variables and classes used by NGSOLVE can be worked with. To use your own methods we have to add the following line at the top of the PDE File.

```
shared = mynavstokelib
```

3.2.1 Integrators

A integrator will either calculate a boundary integral or a region integral. The procedure we have to write, then has to compute the element matrix for each element of the

triangulation. We want to build a new integrator for a divergence term:

$$\int_\Omega \mathrm{div}(bu).$$

For that, we create two files, "myconvection.hpp" and "myconvection.cpp". In the *.hpp file we define, the class, variables and procedures we want to use.

```
namespace ngfem
{
  //build a new class, derived from the BiliniearFormIntegrator class
  class MyConvectionIntegrator : public BilinearFormIntegrator
  {
    //one coefficient should be used
    CoefficientFunction * coef_b;

  //the constructor of the class will get the coefficients
  public:
    MyConvectionIntegrator (const Array<CoefficientFunction*> & coeffs)
      : coef_b(coeffs[0])
    { ; }

    //some variables that will be used

    virtual string Name () const { return "MyDrift"; }
    virtual int DimElement () const { return 2; }
    virtual int DimSpace () const { return 2; }

    //it is not a boundary integral (but a domain integral)
    virtual bool BoundaryForm () const { return false; }

    //procedure that calculates the element matrix
    virtual void CalcElementMatrix (const FiniteElement & fel,
                                    const ElementTransformation & eltrans,
                                    FlatMatrix<double> & elmat,
                                    LocalHeap & lh) const;
  };
}
```

In the *.cpp file we then program what should be done. For the finite element method we now use the basic functions, and get the value of the integral by transforming it to the reference element, so we have to include the Jacobian matrix. For the calculations we have to see that (with a constant b)

$$\mathrm{div}(bu)v = (\frac{\partial b_1 u}{\partial x} + \frac{\partial b_2 u}{\partial y})v = (\nabla u, bv)_{\mathbb{R}}. \tag{38}$$

```
#include <fem.hpp>
#include "myconvection.hpp"

namespace ngfem
{

  /*
    Calculates the element matrix.

    Input is:
```

```
11      the finite element: fel
12      the geometry of the element: eltrans
13
14      Output is:
15      the element matrix: elmat
16
17      Efficient memorymanagement is provided my locheap
18    */
19
20    void MyConvectionIntegrator :: CalcElementMatrix
21                        (const FiniteElement & base_fel,
22                         const ElementTransformation & eltrans,
23                         FlatMatrix<double> & elmat,
24                         LocalHeap & lh) const
25    {
26      /*
27        tell the compiler that we are expecting
28        to get an scalar fe in 2D.
29        if not, an exception will be raised
30      */
31      const ScalarFiniteElement<2> & fel =
32        dynamic_cast<const ScalarFiniteElement<2> &> (base_fel);
33
34      //number of element basis functions:
35      int ndof = fel.GetNDof();
36
37      elmat = 0;
38
39      Matrix<> dshape_ref(ndof, 2); // gradient on reference element
40      Matrix<> dshape(ndof, 2);     // gradient on mapped element
41      Vector<> b_coef(2);
42      Vector<> shape_ref(ndof);  // shape on reference element
43
44      /*
45        get integration rule for element geometry,
46        integration order is 2 times element order
47      */
48      IntegrationRule ir(fel.ElementType(), 2*fel.Order());
49
50      //loop over integration points
51      for (int i = 0 ; i < ir.GetNIP(); i++)
52        {
53          //calculate Jacobi matrix in the integration point
54          MappedIntegrationPoint<2,2> mip(ir[i], eltrans);
55
56          //evaluate the coefficient b
57          coef_b -> Evaluate (mip,b_coef);
58
59          //evaluate the gradient of the basic functions
60          //on the reference element
61          fel.CalcDShape (ir[i], dshape_ref);
62
63          //evaluate the basic functions on the reference element
64          fel.CalcShape(ir[i], shape_ref);
65
66          //transform it to the mapped element
67          dshape = dshape_ref * mip.GetJacobianInverse();
68
```

```
69          // integration weight and Jacobi determinant
70          double fac = mip.IP().Weight() * mip.GetMeasure();
71
72          //elmat_{i,j} +=
73          //fac * InnerProduct(shape_i * b, grad shape_j)
74
75          elmat += fac * shape_ref * Trans(b_coef) * Trans(dshape);
76        }
77    }
78    //register the integrator so it can be used in NGsolve:
79    //name, dimension, coefficients
80    static RegisterBilinearFormIntegrator<MyConvectionIntegrator>
81          initlap ("myconvection", 2, 1);
82  }
```

Now the integrator is finished and can be used in the PDE file by using it for a bilinearform

define bilinearform a -fespace=v
myconvection b

3.2.2 Numerical procedures

Next, new numerical procedures are going to be implemented. As we have seen in Section (3.1) a numerical procedure is called in the PDE file including all bilinear forms and other objects that are used for the calculations. Therefore you have to set them in the constructor of the new procedure.

```
1  #include <solve.hpp>
2
3  using namespace ngsolve;
4
5  //define a new procedure
6
7  class myNumProc : public NumProc
8  {
9          //define objects that are used
10 protected:
11         BilinearForm * bfa;
12         LinearForm * lff;
13         Preconditioner * pre;
14         GridFunction * gfu;
15         NumProc * bvp;
16         double stepend;
17         ...
18
19 public:
20
21   InstationaryNavierStokes (PDE & apde, const Flags & flags)
22     : NumProc (apde)
23   {
24           //set the objects by reading the
25           //information from the *.pde file
26           bfa = pde.GetBilinearForm
27                       (flags.GetStringFlag ("bilinearforma", "a"));
```

```
28            lff = pde.GetLinearForm
29                        (flags.GetStringFlag ("linearform", "f"));
30            pre = pde.GetPreconditioner
31                        (flags.GetStringFlag ("preconditioner", "c"));
32            gfu = pde.GetGridFunction
33                        (flags.GetStringFlag ("gridfunction", "up"));
34            bvp = pde.GetNumProc("np1");
35            stepend = flags.GetNumFlag("stepend", 10);
36            ...
37
38    virtual void Do(LocalHeap & lh)
39          {
40
41          //solve the problem using the objects
42    }
43
44  //register the numerical procedure so it can be used in NGsolve:
45  static RegisterNumProc<InstationaryNavierStokes> npinit1("myNumProc");
```

So every new procedure, first sets the objects by reading the information of the PDE file, and then does the programmed commands of the do-loop. The procedure can then be used by

numproc myNumProc np1 -bilinearform=a -linearform=f -preconditioner=c -gridfunction=up...

3.3 The Navier Stokes equations

3.3.1 Non-linear part

As we have seen in Section (3.1), the terms of a partial differential equation are set in the PDE file by the integrators that stand below the definition. First we have to focus on how to deal with the non-linear part $(u \cdot \nabla)u$ of the stationary Navier Stokes equations

$$\begin{aligned} -\nu\Delta u + \rho(u \cdot \nabla)u + \nabla p &= \rho f \\ \mathrm{div}(u) &= 0 \end{aligned}$$

For that we want to use the so called Oseen iteration: given u^k, find the next iterate (u^{k+1}, p^{k+1}) by solving:

$$\begin{aligned} -\nu\Delta u^{k+1} + \rho(u^k \cdot \nabla)u^{k+1} + \nabla p^{k+1} &= \rho f \\ \mathrm{div}(u^{k+1}) &= 0 \end{aligned} \quad .$$

Under the write conditions this iteration is uniquely solvable. As we have seen in section (1.2) we can write the non-linear part as

$$\rho(u^k \cdot \nabla)u^{k+1} = \rho \mathrm{div} \left(\begin{pmatrix} u_1^{k+1} u^k \\ u_2^{k+1} u^k \end{pmatrix} \right)$$

so we can use our own integrator "myconvection" by using the solution of the last step u^k as vectorfield b. Now the nonlinear part can be solved, and only the Stokes equations are left. See [10].

3.3.2 Stokes equations

For the Stokes equation we will use an integrator provided by NGSOLVE, an so called BDB-integrator. To see this we start with the waek formulation (23) and add both lines to get

$$\int_\Omega \nu \nabla u \nabla v + \int_\Omega \operatorname{div}(v) p + \int_\Omega \operatorname{div}(u) q = \int_\Omega f v \quad \forall (v,q) \in [H^1(\Omega)]^2 \times L^2(\Omega).$$

Together with $\nabla u \cdot \nabla v = \nabla u_x \nabla v_x + \nabla u_y \nabla v_y$ we get

$$\int_\Omega \nu \nabla u_x \nabla v_x + \nabla u_y \nabla v_y + \int_\Omega (\frac{\partial v}{\partial x} + \frac{\partial v}{\partial y}) p + \int_\Omega (\frac{\partial u}{\partial x} + \frac{\partial u}{\partial y}) q$$

or

$$\int_\Omega \underbrace{\begin{pmatrix} \frac{\partial v_x}{\partial x} \\ \frac{\partial v_x}{\partial y} \\ \frac{\partial v_y}{\partial x} \\ \frac{\partial v_y}{\partial y} \\ q \end{pmatrix}^T}_{=:B(v,q)^T} \underbrace{\begin{pmatrix} \nu & 0 & 0 & 0 & 1 \\ 0 & \nu & 0 & 0 & 0 \\ 0 & 0 & \nu & 0 & 0 \\ 0 & 0 & 0 & \nu & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}}_{=:D} \underbrace{\begin{pmatrix} \frac{\partial u_x}{\partial x} \\ \frac{\partial u_x}{\partial y} \\ \frac{\partial u_y}{\partial x} \\ \frac{\partial u_y}{\partial y} \\ p \end{pmatrix}}_{=B(u,p)} = \int_\Omega B^T D B.$$

For the discretization B has to be calculated for every basic function ϕ_i for the velocity components, and ψ_i for the pressure, on all elements. That will produce a matrix.

$$\begin{pmatrix} \frac{\partial \phi_{1,x}}{\partial x} & \dots & \frac{\partial \phi_{nd\_u,x}}{\partial x} & & & & & \\ \frac{\partial \phi_{1,x}}{\partial y} & \dots & \frac{\partial \phi_{nd\_u,x}}{\partial y} & & & & & \\ & & & \frac{\partial \phi_{1,y}}{\partial x} & \dots & \frac{\partial \phi_{nd\_u,y}}{\partial x} & & \\ & & & \frac{\partial \phi_{1,y}}{\partial y} & \dots & \frac{\partial \phi_{nd\_u,y}}{\partial y} & & \\ & & & & & & \psi_1 & \psi_{nd\_p} \end{pmatrix}$$

where *nd_u* and *nd_p* are the dots of freedom. In the procedure of the stokes integrator this is done by the code

```
1  ...
2      //number of dots of freedom for u and p
3      int nd_u = fel_u.GetNDof();
4      int nd_p = fel_p.GetNDof();
5
6      //create a vector with the write size
7      FlatMatrixFixWidth<2> gradu(nd_u, lh);
8
9      //transformation of derivatives from reference element to general element:
10     //evaluate them at the mapped integration points and save them in grad
11     fel_u.CalcMappedDShape (mip, gradu);
12
13     //the shape functions of the pressure
14     FlatVector<> vecp(nd_p, lh);
15     fel_p.CalcShape (mip.IP(), vecp);
16
17     //mat is the matrix where the values should be saved
```

```
18      mat = 0;
19
20      //the first nd_u shape functions belong to u_x,
21      //the next nd_u belong to u_y:
22      mat.Rows(0,2).Cols(cfel.GetRange(0)) = Trans (gradu);
23      mat.Rows(2,4).Cols(cfel.GetRange(1)) = Trans (gradu);
24
25      //and finally nd_p shape functions for the pressure:
26      mat.Row(4).Range(cfel.GetRange(2)) = vecp;
27    }
28  };
29  ...
```

3.3.3 PDE file for the Navier Stokes equations

As everything is now provided we can create the PDE file with the bilinear forms for the Navier Stokes equations. First we will define a finite element space called "stokes" that provides spaces P^k for the components of the velocity and a space P^{k-1} for the pressure as mentioned in section (2.1.4), so the product space $V_h := P^k \times P^k \times P^{k-1}$.

```
define fespace v -stokes -order=2
```

As we will need the solution of the velocity, in our case called "up", for the integrator "myconvection" we have to set this in the PDE file by adding the flag "-addcoef". For the iteration we also have to set start values.

```
define gridfunction up -fespace=v -addcoef
define coefficient ux
0,
define coefficient uy
0,
numproc setvalues npsv1 -gridfunction=up.1 -coefficient=ux
numproc setvalues npsv2 -gridfunction=up.2 -coefficient=uy
define coefficient b
(up.1,up.2),
```

By defining the coefficient "b" including the components of the velocity, we are now able to use it for an integrator. We define the bilinear forms including the parts for the boundary conditions, and as we have a combined finite element space we also have to tell the integrators on which component they act. The mass integrator for the pressure is included with a very small coefficient "reg" to help inverting the matrices as we do not use pivoting

```
define bilinearform a -fespace=v
stokes nu
mass reg -comp=3
myconvection b -comp=1
myconvection b -comp=2
robin penalty -comp=1
```

robin penalty -comp=2

define bilinearform m -fespace=v
mass rho -comp=1
mass rho -comp=2

The bilinear form m is used for the time discretization. Finally the linearform and a direct preconditioner, with the inverting method "pardiso", are defined

define linearform f -fespace=v
neumann ubound -comp=1
neumann ubound2 -comp=2

define preconditioner c -type=direct -bilinearform=a -inverse=pardiso

Now we have to decide what we want to solve, and as we do not have a standard problem, we have to write our own procedures.

3.3.4 Procedure for the stationary Navier Stokes equations

We want to write a new procedure "statNavStoke" to include the iteration for the stationary Navier Stokes equations. To do so we first set a boundary value problem "bvp" in the PDE file that solves the equation with the start values (discussed in Section (3.3.3)). Our new procedure then gets the bilinear form "a", the preconditioner "c" the boundary value problem "bvp" and the number of steps "stepend" for the iteration. The do-loop then computes the method

```
1  virtual void Do(LocalHeap & lh)
2  {
3          for(double n=1;n<stepend;n++)
4          {
5                  //reassemble the matrix A
6                  bfa->ReAssemble(lh);
7                  //update the preconditioner
8                  pre->Update();
9                  //solve the problem again with the updated coefficients
10                 bvp->Do(lh);
11                 //tell NGsolve to redraw the solution
12                 Ng_Redraw();
13         }
14 }
```

Now the procedure can be used, so we add the following in our PDE file

numproc bvp np1 -bilinearform=a -linearform=f -gridfunction=up
-preconditioner=c -maxsteps=100

numproc statNavStoke np2 -bilinearform=a -linearform=f
-gridfunction=up -preconditioner=c -stepend=15

3.3.5 Procedure for the in stationary Navier Stokes equations

As we have seen, we need an iteration for the non-linear part, therefore we have to be careful when using a time discretization. We start with the equation (32)

$$M\frac{u_{j+1}-u_j}{\tau}+A(u_{j+1})=f_{j+1}$$

with $A(u_{j+1}) := A_{u_{j+1}}u_{j+1}$. The matrix $A_{u_{j+1}}$ represents the solution of the iteration with u_j^0, so the solution of the last time-step, as starting value. Using the implicit Euler method we get

$$Mu_{j+1}^{k+1}+\tau A_{u_{j+1}^k}u_{j+1}^{k+1}=\tau f_{j+1}+Mu_j^0$$

or

$$u_{j+1}^{k+1}=(M+\tau A_{u_{j+1}^k})^{-1}(\tau f_{j+1}+Mu_j^0)\quad \text{for } k=0,\dots,k_{max},$$

and set the solution of the time-step $u_{j+1}=u_{j+1}^{k_{max}}$. To include this method we make a new numerical procedure "instatNavStoke". The procedure should use the bilinear form "a" for the stationary problem, bilinear form "m" for the mass matrix, the time interval "dt" , the end time "tend", and the number of steps "stepend" that should be used for the iteration. The time discretization then again is implemented in the do-loop of the procedure including the vectors

$$d(u_j) := (\tau f_{j+1}+Mu_j)$$
$$w := (M+\tau A_{u_{j+1}^k})^{-1}.$$

Also we assume that the right side f is stationary, so we do not have to update this vector.

```
1  ...
2    virtual void Do(LocalHeap & lh)
3          {
4                  //bilinearform a
5                  BaseMatrix & mata = bfa->GetMatrix();
6                  //bilinearform m
7                  const BaseMatrix & matm = bfm->GetMatrix();
8                  //linearform f
9                  BaseVector & vecf = lff->GetVector();
10                 //the solution u
11                 BaseVector & vecu = gfu->GetVector();
12
13                 //creates a matrix of same type:
14                 BaseMatrix & summat = *matm.CreateMatrix();
15
16                 //create additional vectors and matrices:
17                 BaseVector & d = *vecu.CreateVector();
18                 BaseVector & w = *vecu.CreateVector();
19
20                 BaseMatrix & invmat = * summat.InverseMatrix
21                                 (gfu->GetFESpace().GetFreeDofs());
22
23                 //set the type of the inversion
24                 summat.SetInverseType(PARDISO);
25
26                 for(double t=0; t <= tend; t+=dt)
27                 {
```

```
28                              //next time step
29                              //use old solution for start value of iteration
30                              d = dt*vecf+matm*vecu;
31
32                              for(double n=1;n<=stepend;n++)
33                              {
34                                      //step of iteration
35                                      bfa->ReAssemble(lh);
36                                      summat.AsVector() = matm.AsVector() +
37                                      dt* bfa->GetMatrix().AsVector();
38                                      BaseMatrix & invmat = * summat.InverseMatrix
39                                              (gfu->GetFESpace().GetFreeDofs());
40                                      w=invmat*d;
41                                      //set new solution
42                                      vecu=w;
43                                      delete & invmat;
44                              }
45                              //tell NGsolve to redraw the solution
46                              Ng_Redraw();
47                      }
48      }
49  ...
```

Now we can use the procedure, so we add the following in the PDE file for solving the in stationary Navier Stokes equations with $\tau = 0.01, T = 10, k_{max} = 5$, and using the implicit Euler method.

numproc instatNavStoke np2 -bilinearform=a -bilinearform=m -linearform=f -gridfunction=up -preconditioner=c -stepend=5 - tend=10 -dt=0.01 -method=IE

When we want to use the DIRK method, we have to make two iterations, one for k_1 and one for k_2. We start with the iteration for k_1, which can be seen as an implicit Euler method with step size $\tau\alpha$. Looking at (36) and using s for the iteration index we get

$$k_1^{s+1} = (M + A_{u^s_{j+\alpha}}\tau\alpha)^{-1}(f_{j+\alpha} - A_{u^s_{j+\alpha}}u_j^0)$$
$$u_{j+\alpha}^{s+1} = u_j^0 + \tau\alpha k_1^{s+1}.$$

with starting value u_j^0, and set $u_{j+\alpha} = u_{j+\alpha}^{s_{max}}$. For the second iteration we now use $k_1 = k_1^{s_{max}}$ to get

$$k_2^{s+1} = (M + A_{u^s_{j+1}}\tau\alpha)^{-1}(f_{j+1} - A_{u^s_{j+1}}(u_j^0 + \tau(1-\alpha)k_1)$$
$$u_{j+1}^{s+1} = u_j^0 + \tau \underbrace{b_1}_{=1-\alpha} k_1 + \tau \underbrace{b_2}_{\alpha} k_2^{s+1}.$$

```
1   ...
2   double alpha = 1-sqrt(0.5);
3   cout<<alpha<<endl;
4   for(double t=0; t <= tend; t+=dt)
5   {
6           //k1
7           v=vecu; //start value u_{j}
8           for(double n=1;n<=stepend;n++)
9           {
```

```
10                  bfa->ReAssemble(lh);
11                  summat.AsVector() = matm.AsVector() +
12                          dt*alpha* bfa->GetMatrix().AsVector();
13                  BaseMatrix & invmat = * summat.InverseMatrix
14                          (gfu->GetFESpace().GetFreeDofs());
15                  r=vecf-bfa->GetMatrix()*v;
16                  k1=invmat*r;
17                  vecu=v+dt*alpha*k1;
18                  delete & invmat;
19          }
20          //k2
21          v += dt*(1-alpha)*k1;
22          for(double n=1;n<=stepend;n++)
23          {
24                  bfa->ReAssemble(lh);
25                  summat.AsVector() = matm.AsVector() +
26                          dt*alpha* bfa->GetMatrix().AsVector();
27                  BaseMatrix & invmat = * summat.InverseMatrix
28                          (gfu->GetFESpace().GetFreeDofs());
29                  r=vecf-bfa->GetMatrix()*v;
30                  k2=invmat*r;
31                  vecu=v+dt*alpha*k2;
32                  delete & invmat;
33          }
34          Ng_Redraw();
35  }
36  ...
```

And include the same as before, but change the method

numproc instatNavStoke np2 -bilinearform=a -bilinearform=m -linearform=f -gridfunction=up -preconditioner=c -stepend=5 - tend=10 -dt=0.01 -method=DIRK

3.4 Natural convection

As discussed in Section (1.3) the natural convection problem is a coupled problem of the Navier Stokes equations and the convection diffusion equation for the temperature. The idea is to first compute the velocity to use it in the next step to update the temperature, make a time step and start the cycle again.

3.4.1 PDE file for the natural convection

To realize this coupling we have to be able to use the temperature as a coefficient on the right side of the Navier Stokes equations. We start with the PDE file of the Navier Stokes equations and first add a new finite element space and a grid function for the temperature with a start value "TI".

define fespace vt -type=h1ho -order=2
define gridfunction T -fespace=vt -addcoef

numproc setvalues stemp -gridfunction=T -coefficient=TI

Now we can use "T" and "up" as coefficients, so we add a new bilinear form "B" and a linear form "g" for the temperature, and a linearform "f" for the Navier Stokes equations. Also a bilinear form "m2" for the mass matrix of the temperature is created, beacuase we will need it for the time discretization.

```
...
define coefficient uboundnav
(-9.81*(1-beta*(T-T0))),
...

define bilinearform B -fespace=vt
laplace lam
myconvection b
robin penalty

define bilinearform m2 -fespace=vt
mass rho

define linearform f -fespace=v
neumann ubound2 -comp=1
neumann ubound2 -comp=2
source uboundnav -comp=2

define linearform g -fespace=vt
source temp_coef
neumann ubound
```

The coefficients for the integrators have to be defined as described in the problem.

3.4.2 Procedure for the coupled problem

To solve the coupled problem, we write a new numerical procedure "NatConv". For both steps we want to use the implicit Euler method, so in the do-loop we first make a step for the velocity, including the iteration, and then a step for the temperature, were we use the normal implicit Euler method as discussed in section (2.3.1).

```
1  virtual void Do(LocalHeap & lh)
2  {
3          for(double t=0; t <= tend; t+=dt)
4          {
5                  //assemble left side of the NVS with new temperature
6                  lff->Assemble(lh);
7                  //time step for NVS
8                  d = dt*(lff->GetVector())+matm*vecu;
9                  for(double n=1;n<=stepend;n++)
10                 {
11                         bfa->ReAssemble(lh);
```

```
12                          summat.AsVector() = matm.AsVector() +
13                                              dt* bfa->GetMatrix().AsVector();
14                          BaseMatrix & invmat = * summat.InverseMatrix
15                                              (gfu->GetFESpace().GetFreeDofs());
16                          w=invmat*d;
17                          vecu=w;
18                          delete & invmat;
19                      }
20
21                      //time step temperature
22                      bfb->ReAssemble(lh);
23                      summat2.AsVector() = (1.0/dt) * matm2.AsVector() +
24                                              matb.AsVector();
25                      BaseMatrix & invmat2 = * summat2.InverseMatrix
26                                              (gft->GetFESpace().GetFreeDofs());
27                      d2 =  vecg - matb * vect;
28                      w2 = invmat2 * d2;
29                      vect += w2;
30                      delete & invmat2;
31                      Ng_Redraw();
32              }
33      }
```

To include the method, we add the following in the PDE file

numproc NatConv np1 -bilinearforma=a -bilinearformm=m -bilinearformb=B -bilinearformm2=m2 -linearformf=f -linearformg=g -gridfunctionu=up -grudfunctiont=T -preconditioner=c -stepend=5 -tend=10 -dt=0.01

4 Examples

4.1 Laminar flow around a cylinder

The first example is a benchmark computation of a laminar flow around a cylinder given by [8]. We want to use the given geometry and coefficients, and compare the results with different polynomial degrees k.

4.1.1 Fluid properties

For the examples the fluid properties will all be the same

$$\begin{aligned} \rho\frac{\partial u}{\partial t} - \nu\Delta u + \rho(u\cdot\nabla)u + \nabla p &= 0 \\ \operatorname{div}(u) &= 0 \end{aligned} \tag{39}$$

with the density $\rho = 1.0$ kg/m^3 and the kinematic viscosity $\nu = 10^{-3}$ m^2/s.

4.1.2 Geometry

The geometry is given in Figure (3), with $H = 0.41$ as the height and $D = 0.1$ as the diameter of the circle. We would like to have the following boundary conditions for the

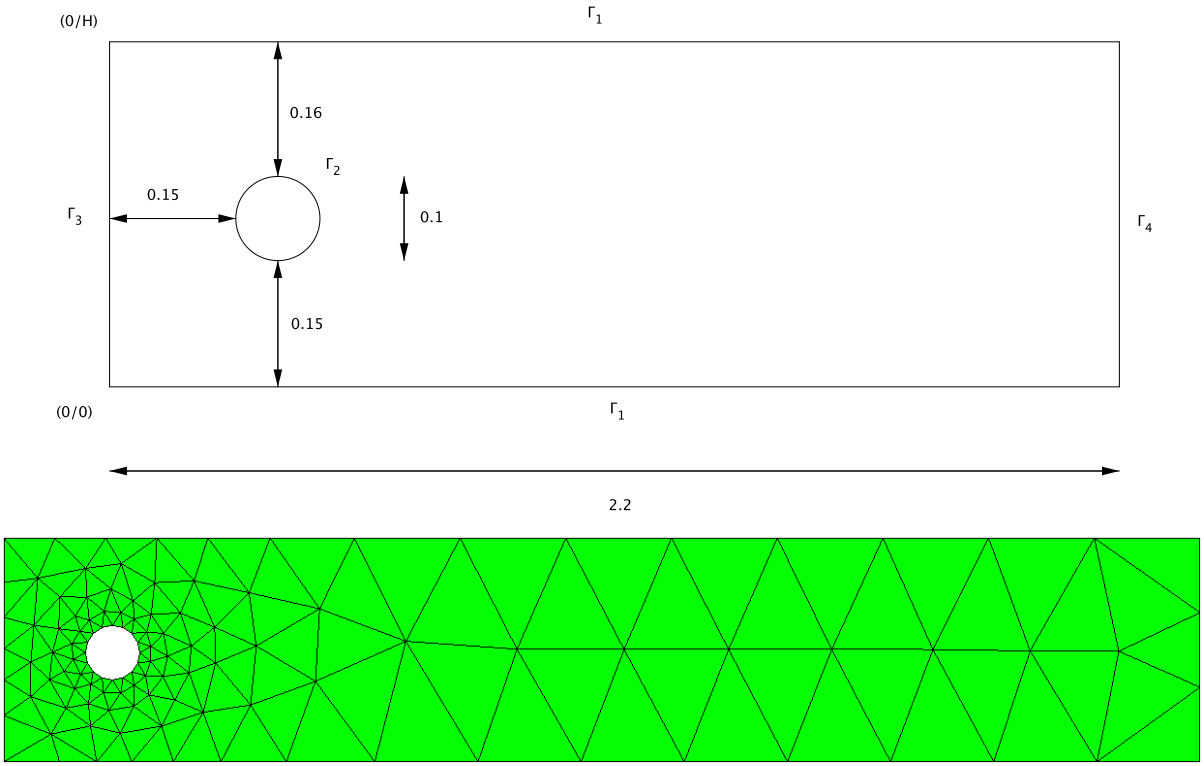

Figure 3: Geometry and mesh for the laminar flow test case

Navier Stokes equations

$$\begin{aligned} u &= (0,0)^T \quad \text{on } \Gamma_1 \cup \Gamma_2 \times [0,T] \\ u &= u_D \qquad \text{on } \Gamma_3 \times [0,T] \end{aligned}$$

with u_D as the inflow value, and Γ_4 as the outflow boundary. We save the geometry in the file "rechole.in2d", generate a mesh with NetGen, and save it in the file "rechole.vol".

4.1.3 Computed values

To compare the results we compute the drag and the lift coefficients. Therefore we first need the drag and lift forces

$$F_D = \int_{\Gamma_2} (\nu \frac{\partial u_t}{\partial n} n_y - p n_x)\, ds \qquad F_L = -\int_{\Gamma_2} (\nu \frac{\partial u_t}{\partial n} n_x + p n_y)\, ds \tag{40}$$

with u_t as the scalar tangential velocity on Γ_2, and n as the normal vector on Γ_2. The coefficients then are given by

$$c_D = \frac{2F_D}{\rho \overline{u}^2 D} \qquad c_D = \frac{2F_L}{\rho \overline{u}^2 D}$$

with

$$\overline{u}(t) := \frac{2u_D(0, H/2, t)}{3}.$$

To compute the forces we look back at the weak formulation of the Stokes equations, as the non-linear convection term won't take effect on Γ_2, and include the terms that occur by using Robin penalties

$$\begin{aligned}\int_\Omega \nu\nabla u\nabla v + \int_\Omega \operatorname{div}(v)p + \alpha\int_{\partial\Omega}\nu uv &= \alpha\int_{\partial\Omega}\nu u_D v\\ \int_\Omega \operatorname{div}(u)q &= 0.\end{aligned}$$

Now we integrate back in the first line, to get the strong formulation

$$\int_\Omega -\nu\Delta u\nabla v - \int_{\partial\Omega}\nu(\nabla u\cdot n)v - \int_\Omega \nabla pv + \int_{\partial\Omega} p(v\cdot n) = \alpha\int_{\partial\Omega}\nu uv + \alpha\int_{\partial\Omega}\nu u_D v.$$

So the Stokes equations

$$\int_\Omega -\nu\Delta u\nabla v - \int_\Omega \nabla pv = 0$$

and

$$\alpha\int_{\partial\Omega}\nu uv + \alpha\int_{\partial\Omega}\nu u_D v = \int_{\partial\Omega}\nu(\nabla u\cdot n)v - \int_{\partial\Omega} p(v\cdot n).$$

The right side is the sum of our forces F_D and F_L. To see this, consider that we can write $u = (u_\tau, u_n)$ or $u = u_T + u_N$ with $u_T = \tau(u\cdot\tau)$ and $u_T = n(u\cdot n)$, so we get

$$\begin{aligned}\int_{\partial\Omega}\nu(\nabla u\cdot n)v - \int_{\partial\Omega} p(v\cdot n) &= \int_{\partial\Omega}\nu\frac{\partial u}{\partial n}v - \int_{\partial\Omega} p(v\cdot n) =\\ &= \int_{\partial\Omega}\nu\left(\frac{\partial u_T}{\partial n} + \frac{\partial u_N}{\partial n}\right)v - \int_{\partial\Omega} p(v\cdot n).\end{aligned}$$

We now look at the continuity equation that can be written as

$$\operatorname{div}(u) = \frac{\partial u_\tau}{\partial\tau} + \frac{\partial u_n}{\partial n} = 0,$$

and as $u_\tau = 0$ on Γ_2 also

$$\frac{\partial u_\tau}{\partial\tau} = 0,$$

and we get

$$\frac{\partial u_n}{\partial n} = 0 \quad\text{and}\quad \frac{\partial u_N}{\partial n} = (0,0)^T.$$

For the right side we now have

$$\begin{aligned}\int_{\partial\Omega}\nu\left(\frac{\partial u_T}{\partial n} + \frac{\partial u_N}{\partial n}\right)v - \int_{\partial\Omega} p(v\cdot n) &= \int_{\partial\Omega}\nu\frac{\partial u_T}{\partial n}v - \int_{\partial\Omega} p(v\cdot n)\\ &= \int_{\partial\Omega}\nu\frac{\partial u_\tau}{\partial n}\tau v - \int_{\partial\Omega} p(v\cdot n),\end{aligned}$$

and together with $\tau = (n_y, -n_x)^T$, and just considering Γ_2 with $u_D = (0, 0)$, we finally get

$$\alpha \int_{\Gamma_2} \nu uv = \int_{\Gamma_2} \nu \frac{\partial u_\tau}{\partial n} \tau v - \int_{\Gamma_2} p(v \cdot n) =$$
$$= \int_{\Gamma_2} (\nu \frac{\partial u_t}{\partial n} n_y - p n_x) - \int_{\Gamma_2} (\nu \frac{\partial u_t}{\partial n} n_x + p n_y) = F_D + F_L.$$

To compute the values we just have to add two linear forms with the write value α so only Γ_2 is included

$$f_D(v) := \int_{\partial\Omega} \alpha \nu v_1 \quad \text{and} \quad f_L(v) := \int_{\partial\Omega} \alpha \nu v_2.$$

In the PDE file we define the coefficient "force"

define coefficient force
0, 1e10, 0, 0,

define linearform fD -fespace=v
neumann force -comp=1

define linearform fL -fespace=v
neumann force -comp=2

To get the valus for F_D and F_L, we then only have to add the following lines in our numerical procedures

```
1  ...
2          double FL,FD;
3  ...
4          lfFL = pde.GetLinearForm (flags.GetStringFlag ("forceFL", "fL"));
5          lfFD = pde.GetLinearForm (flags.GetStringFlag ("forceFD", "fD"));
6  ...
7          BaseVector & vecFL = lfFL->GetVector();
8          BaseVector & vecFD = lfFD->GetVector();
9  ...
10         FC=InnerProduct(vecFL, vecu);
11         FD=InnerProduct(vecFD, vecu);
12 ...
```

4.1.4 Test case 1 - stationary

The first test case will be a stationary problem. The inflow condition is time independent and set as

$$u_D(x, y, t) = \left(\frac{4 u_m y(H - y)}{H^2}, 0 \right)^T$$

with $u_m = 0.3$ m/s. For the computation we want to set the polynomial degree $k = 5$ and use different numbers of iterations *stepend*. We set the coefficient "ubound" as

define coefficient ubound
0, 0, (1e10*4*0.3*y*(0.41-y)/0.1681),0

including $\alpha = 10^{10}$, and start the numerical procedure including the linear forms for the forces

numproc statNavStoke np2 -bilinearform=a -linearform=f -gridfunction=up -preconditioner=c -forceFD=fD -forceFL=fL -stepend=15

For the two coefficients we compare the results (Table (3)) with the values we get using *stepend* = 15

$$c_{D_{ref}} = 5.50607 \quad \text{and} \quad c_{L_{ref}} = 0.00984321.$$

stepend	c_D	c_L	$c_D - c_{D_{ref}}$	$c_L - c_{L_{ref}}$
1	3.093900	0.029417	2.412170	-0.019573
2	5.493330	0.158336	0.012740	-0.148493
3	5.475900	0.022907	0.030170	-0.013064
4	5.499830	0.015446	0.006240	-0.005603
5	5.510060	0.011283	-0.003990	-0.001440
6	5.508480	0.010180	-0.002410	-0.000337
7	5.507010	0.009736	-0.000940	0.000107
8	5.506360	0.009832	-0.000290	0.000011
9	5.506150	0.009859	-0.000080	-0.000016
10	5.506090	0.009838	-0.000020	0.000005
11	5.506080	0.009841	-0.000010	0.000002
12	5.506080	0.009844	-0.000010	-0.000001
13	5.506070	0.009843	0.000000	0.000000
14	5.506070	0.009843	0.000000	0.000000

Table 3: Drag and lift coefficient for *stepend* = 1 . . . 15

It can be seen, that the solution converges by increasing the numbers of iterations. The stationary solution of the velocity and the pressure can be seen in Figure (4).

4.1.5 Test case 2 - unsteady

Next we want to look at an non-stationary problem. The inflow condition will still be time independent and is set as

$$u_D(x, y, t) = \left(\frac{4u_m y(H - y)}{H^2}, 0 \right)^T$$

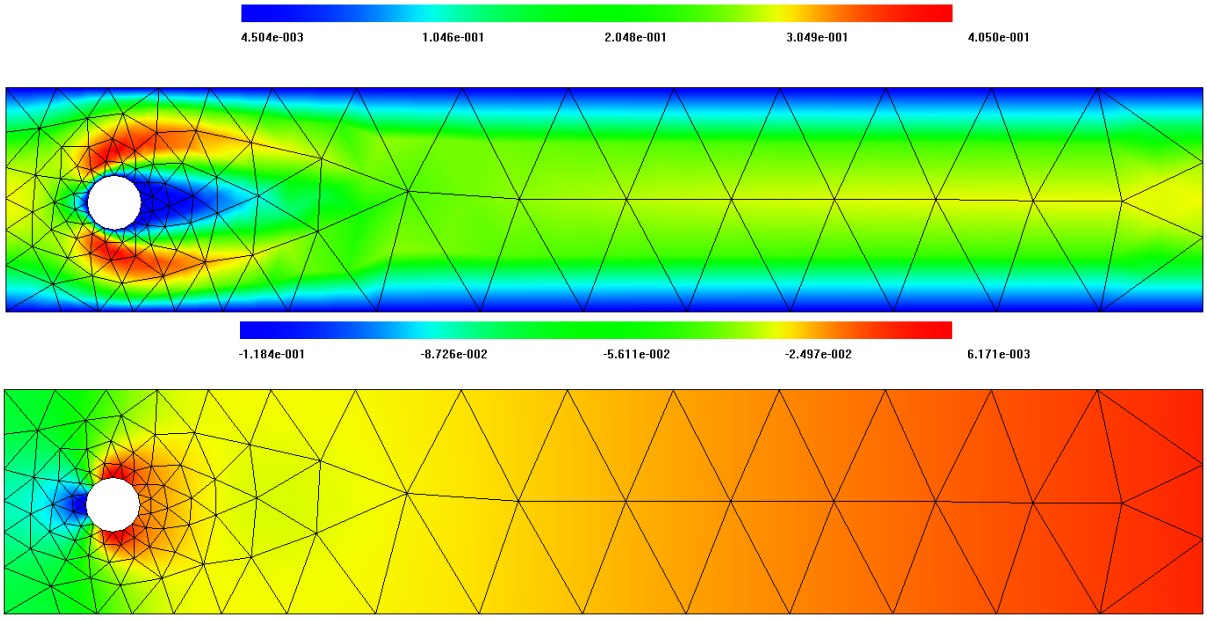

Figure 4: Stationary solution of the velocity and pressure with $k = 15$ and $stepend = 15$

with $u_m = 1.5$ m/s. So we set the coefficient "ubound" as

define coefficient ubound
0, 0, (1e10*4*1.5*y*(0.41-y)/0.1681),0

To compare the results the number of iterations is set on $stepend = 5$ and the end time is $tend = 10$. We will then change the number of the polynomial degree k and compare the results of c_D and c_L. For the time discretization we now have two options. We can either use the implicit Euler method or the DIRK method. For the first one, we choose the time step $dt = 0.005$, and for the second one $dt = 0.01$. Although we use the doubled time step, the DIRK method delivers better results, as it is a second order method. We then call our numerical procedures by either

numproc instatNavStoke np2 -bilinearform=a -bilinearform=m
-linearform=f -gridfunction=up -preconditioner=c
-stepend=5 -tend=10 -dt=0.005 -forceFD=fD -forceFL=fL
-method=IE

or

numproc instatNavStoke np2 -bilinearform=a -bilinearform=m
-linearform=f -gridfunction=up -preconditioner=c
-stepend=5 -tend=10 -dt=0.01 -forceFD=fD -forceFL=fL
-method=DIRK

In Figure (5) it can be seen, that it takes some time until the coefficients and the solution converges. Therefore we use the maximum values of c_D and c_L in one period after convergence is given. The results then are compared in Table (4) with the reference

values given by the simulation using the DIRK method and $k = 5$.

$$c_{D_{ref}} = 3.187430 \quad \text{and} \quad c_{L_{ref}} = 0.969043.$$

As mentioned before, it can be seen, that even with $k = 5$ the implicit Euler method is

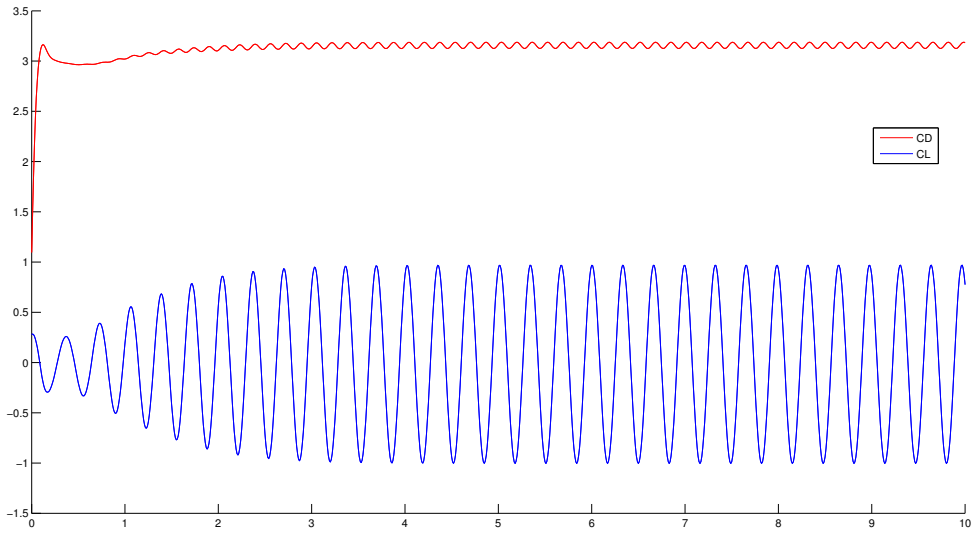

Figure 5: Drag coefficient c_D and lift coefficient c_L for $k = 5$ and using the DIRK method

k	$c_{D_{max}}$	$c_{L_{max}}$	$c_{D_{max}} - c_{D_{ref}}$	$c_{L_{max}} - c_{L_{ref}}$	method
2	3.140990	0.986291	0.046440	-0.017248	DIRK
2	3.001940	0.619345	0.185490	0.349698	IE
3	3.223330	1.050610	-0.035900	-0.081567	DIRK
3	3.047500	0.508779	0.139930	0.460264	IE
4	3.184110	0.967735	0.003320	0.001308	DIRK
4	3.039130	0.524872	0.148300	0.444171	IE
5	3.041150	0.517827	0.146280	0.451216	IE

Table 4: Drag coefficient, lift coefficient and error for $k = 1 \dots 5$

not as good as using the DIRK method with only $k = 2$. In Figure (6) and (7) a snapshot at $t = 5$s was taken.

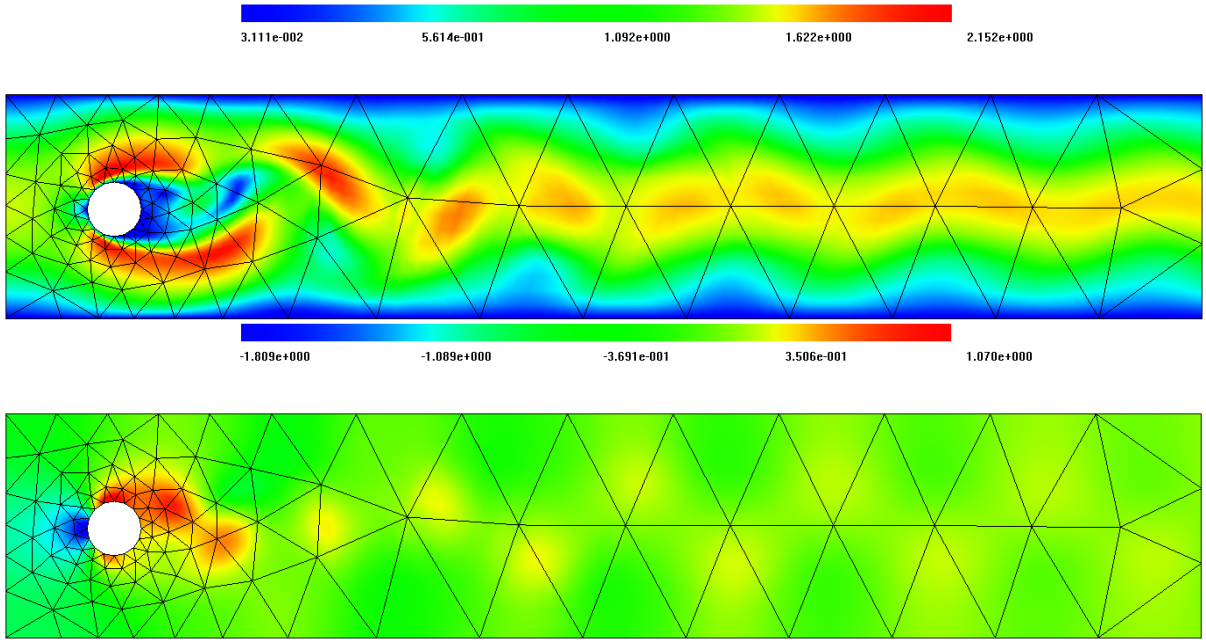

Figure 6: Absolute value of the velocity and value of pressure, $k = 5, t = 5$ s, $stepend = 5, dt = 0.01$

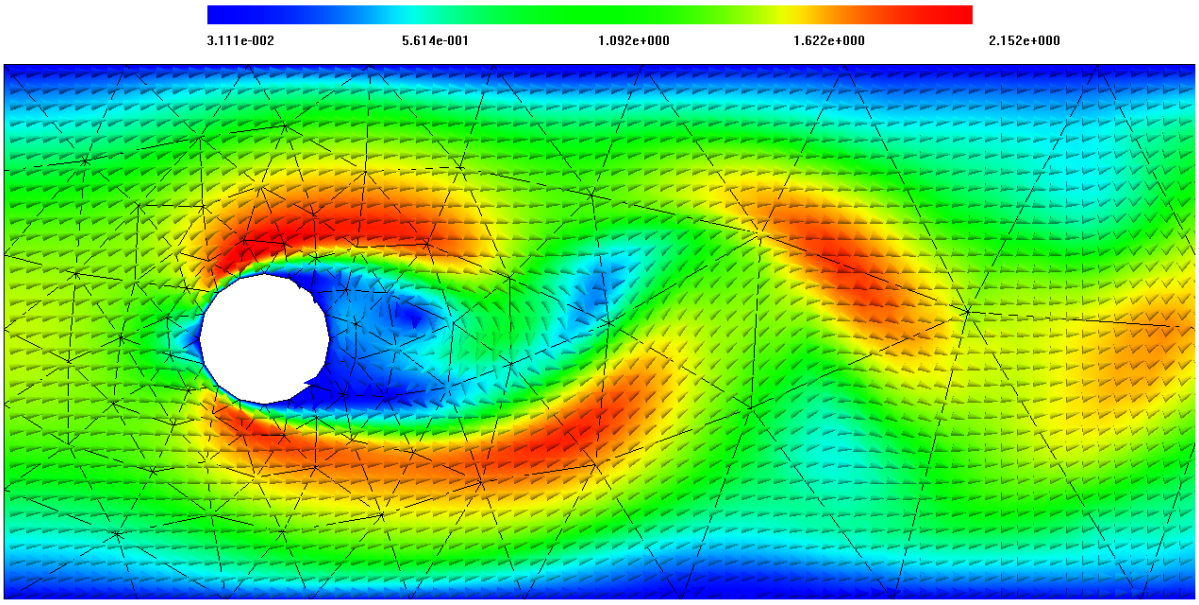

Figure 7: Absolute value and direction of the velocity near the cylinder, $k = 5, t = 5$ s, $stepend = 5, dt = 0.01$

4.2 Rayleigh-Benard convection

The Rayleigh-Benard convection, is a natural convection that occurs in a horizontal layer of fluid that is heated from below, in which the fluid develops regular convection cells. We want to take the geometry and coefficients used in tutorial 10 in [5], and compare the results.

4.2.1 Fluid and heat properties

For the fluid and the temperature we use the equations discussed in (1.3)

$$\begin{aligned} \frac{\partial u}{\partial t} - \frac{\nu}{\rho_0}\Delta u + (u\cdot\nabla)u + \frac{\nabla p}{\rho_0} &= g(1-\beta(T-T_0)) \\ \mathrm{div}(u) &= 0 \end{aligned}$$

and

$$\frac{\partial T}{\partial t} - \lambda\Delta T + \mathrm{div}(uT) = 0.$$

with the following coefficients

parameter	variable	value
reference density	ρ_0	998.3 kg/m^3
viscosity	ν	$1040e^{-6}$ Ns/m^2
heat capacity	C	4183 J/(kgK)
heat conductivity	κ	0.58 W/(mK)
heat expansion coefficient	β	$2.07e^{-4}$ K^{-1}
reference temperature	T_0	293 K

So we set the coefficients in the PDE file as

$$\frac{\nu}{\rho_0} = 1.04177e^{-6} \quad \text{and} \quad \lambda = \frac{\kappa}{C\rho} = 1.3889e^{-7}$$

by

define coefficient nu
(1.04177*1e-6),
define coefficient lam
(1.3889*1e-7),
define coefficient beta
(2.08*1e-4),
define coefficient T0
293,

Note that the solution of the pressure p will be scaled by ρ_0.

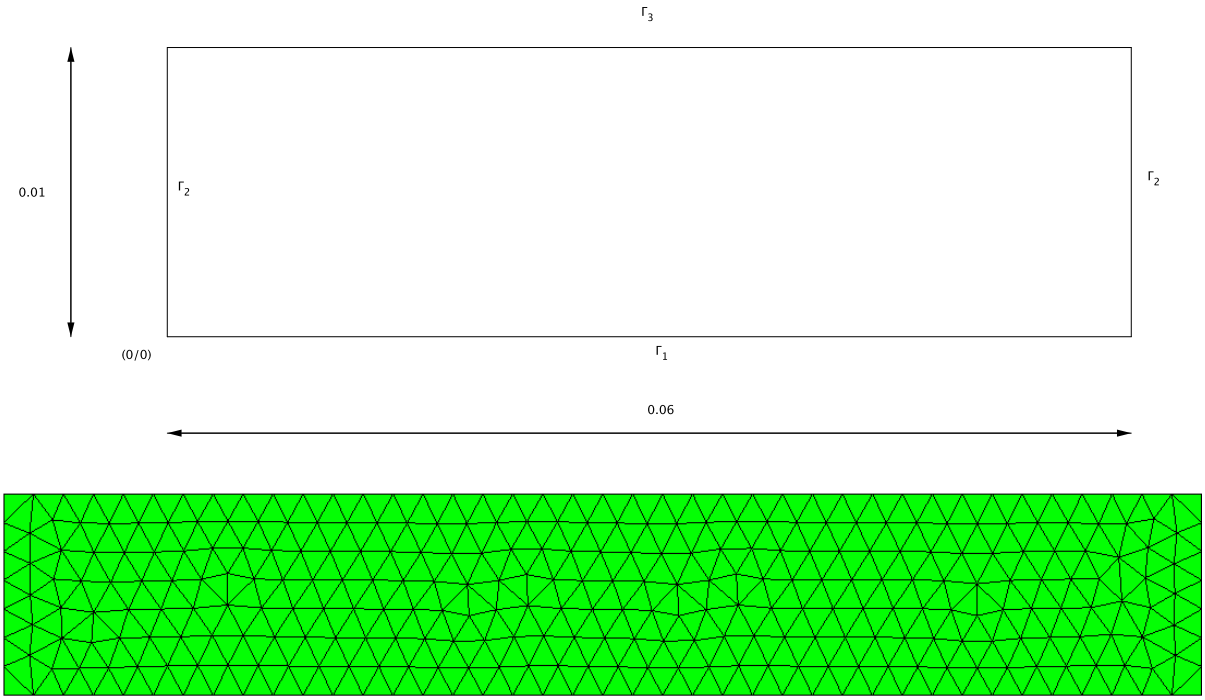

Figure 8: Geometry and mesh of the natural convection test case

4.2.2 Geometry

The geometry is given by figure (8), and the boundary conditions are set as

$$\begin{aligned} u &= (0,0)^T \quad \text{on } \Gamma_1 \cup \Gamma_2 \cup \Gamma_3 \times [0,T] \\ T &= 293.5 \text{ K} \quad \text{on } \Gamma_1 \times [0,T] \\ T &= 293 \text{ K} \quad \text{on } \Gamma_3 \times [0,T] \end{aligned}$$

We use NetGen to create a mesh and save it as "rec.in2d" and "rec.vol". For the boundary conditions the coefficients "penalty" and "ubound" are set for the temperature, and "penalty2" and "ubound2" are set for the velocity.

define coefficient penalty
1e10, 0, 1e10

define coefficient penalty2
1e10,1e10,1e10,

define coefficient ubound
(1e10*293.5), 0, (1e10*293)

define coefficient ubound2
0,0,0,

4.2.3 Solution

Before we can use our procedure for the natural convection problem, we first have to define the coefficient "uboundnav"(see (3.4.1)). In the definition of the linear form for the Navier Stokes equations in the PDE file, we see that the source integrator just acts on the second component, as the gravity force only acts from top to bottom. So we just have to add

define coefficient uboundnav
(-9.81*(1-beta*(T-T0))),

As there are two stationary solutions, we will start the simulation with a small numerical noise, like giving a little push to a pendulum that stands on top to bring it to the stationary solution at the bottom. When looked at the solution provided by [5], we see that there should occur 7 convections cells. The numerical noise is a sine wave with a small amplitude set with the numerical procedure "setvalues". Consider that the start value is not in conflict with the boundary conditions.

define coefficient TI
(293.5-50*y+y*(0.01-y)*0.1*1e4*sin(20/0.06*x*pi)),

numproc setvalues stemp -gridfunction=T -coefficient=TI

As the temperature is not linear our solution already shows convections cells at $t = 0$. When time passes, it can be seen that some of the convection cells combine together and finally end up in the stationary solution, see figure (9)(10) and (11).

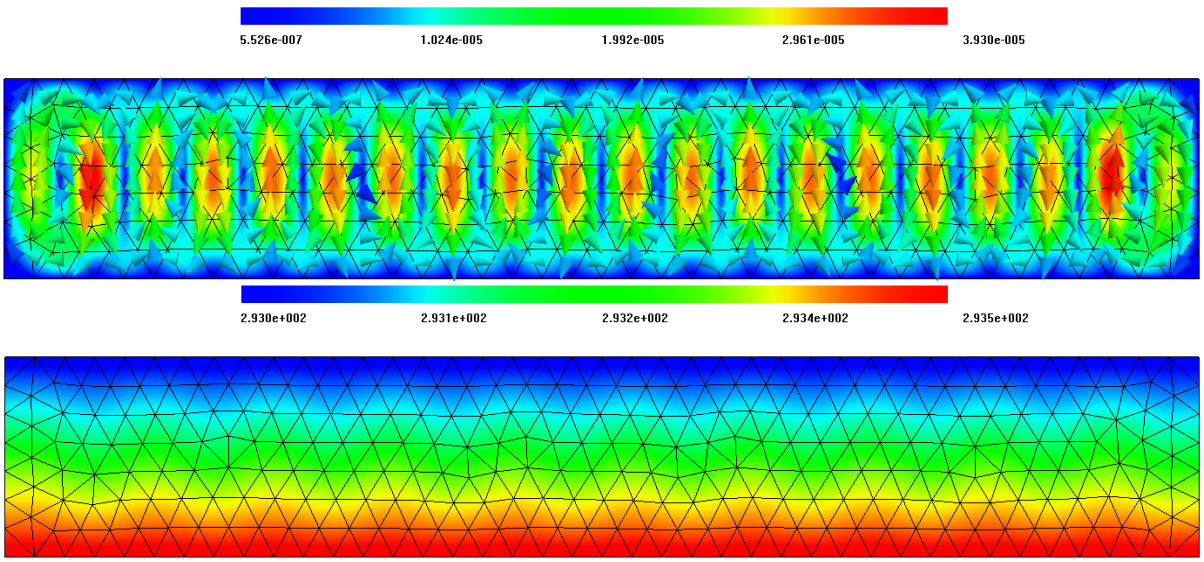

Figure 9: Velocity and temperature at $t = 0$ s

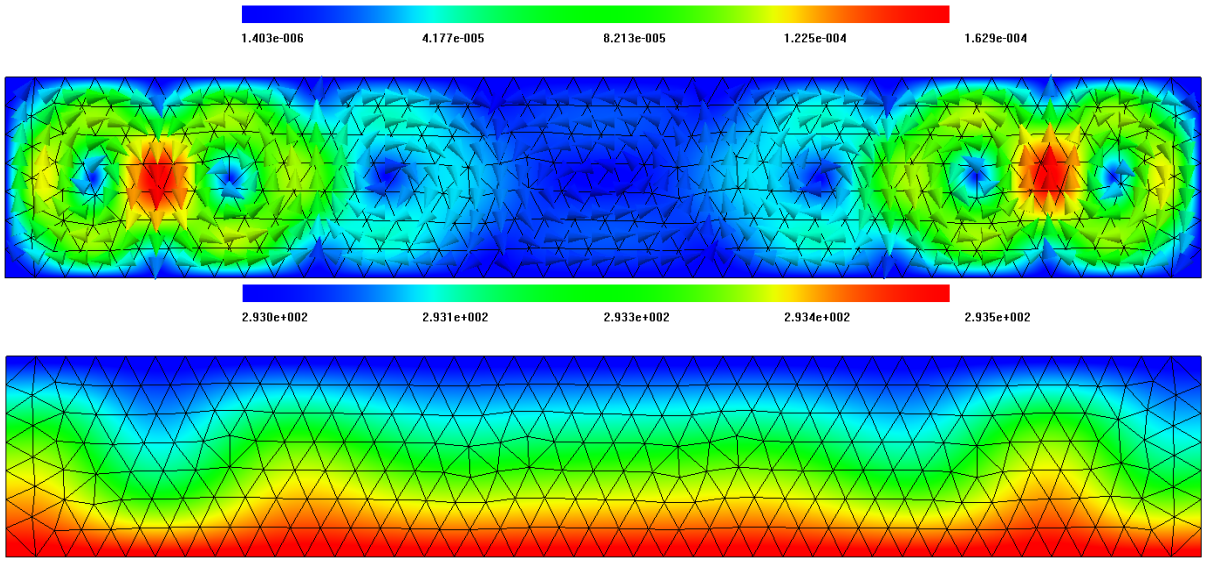

Figure 10: Velocity and temperature at $t = 100$ s

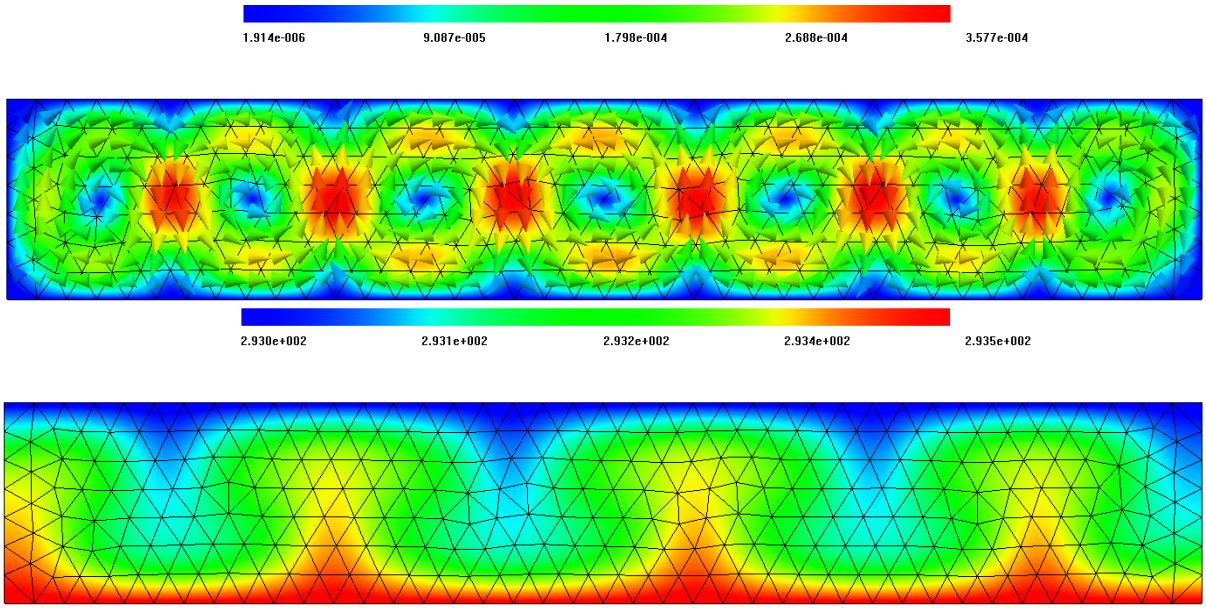

Figure 11: Velocity and temperature at $t = 500$ s

References

[1] Dietrich Braess. *Finite Elemente*. Springer, New York, 2000.

[2] Alexandre J. Chorin and Jerrold E. Marsden. *A Mathematical Introduction to Fluid Mechanics*. Springer, New York, 1992.

[3] Jean Donea and Antonio Huerta. *Fenite Element Methods for Flow Problems*. Wiley, 2003.

[4] Gerhard Wanner Ernst Hairer, Syvert P. Norsett. *Solving Ordinary Differential Equations I*. Springer, 1993.

[5] CSC IT Center for Science. Elmer tutorials.

[6] Michel Fortin Franco Brezzi. *Mixed and Hybrid Finite Element Methods*. Springer, New York, 1991.

[7] Univ.-Prof. Dr. Ansgar Jüngel. *Partielle Differentialgleichungen*. 2013.

[8] S. Turek M. Schäfer. Benchmark computations of laminar flow around a cylinder.

[9] Prof. Dr. Joachim Schöberl. *Elementare Numerische Methoden für partielle Differentialgleichungen*. 2009.

[10] Prof. Dr. Joachim Schöberl. *Numerical Methods for Partial Differential Equations*. 2009.