Implementation of a Stationary Navier Stokes Equation Solver

Florian Reichl, Oliver Meister

September 21, 2007

Contents

1 Introduction

At the ”Lehrstuhl Informatik 5” at the TU München, a research project in the field of efficient fluid simulation called ”Peano” is being developed. Peano concentrates on incompressible flows in two and three dimensions and uses a finite element method for the approximation of the solution to a problem. The API PETSc (Portable Extensible Toolkit for Scientific Computation) is used for that purpose.

Subject of this project is the implementation of a solver which has the special task of finding those steady state solutions by solving the (discrete) Navier-Stokes Equation implicitly. Therefore a wrapper is needed to link the non-linear functionality of PETSc to Peano.

The project itself was divided into seven steps:

1. **Familiarization with the problem:**
   The first step was getting familiar with solving non-linear equation systems. This includes collecting several equations which can be used to test a solver's behavior in some selected cases.

2. **Testing PETSc:**
   The test equations collected in step one should be implemented using PETSc. Several solution methods are to be tested and compared to each other.

3. **Wrapping all necessary PETSc methods:**
   The necessary PETSc methods are wrapped into classes and integrated into the Peano project. These classes have to be consistent with the Peano coding standards.

4. **Setting up the Navier-Stokes equations:**
   The continuous stationary Navier-Stokes equations are to be implemented in a discretized form on a regular grid in 2D.

5. **Solving several test scenarios:**
   Several test scenarios should be implemented, documented and compared to each other.

6. **Alternative calculation of the Jacobian matrix:**
   If necessary, the calculation of the Jacobian matrix should be optimized.

7. **Linking the calculation to the time step based solver:**
   The new solver is linked to the existing one to improve the initial value for better performance.

2 Preparation

2.1 Installation

The first step of the project was getting a windows computer ready to compile Peano. We used Cygwin 1.5.24-2 on Windows XP SP2 with the gcc 3.4.4 compiler. Make, Python and gcc containing g++ and g77 have to be installed.
After that, we created two different PETSc builds for debug and release purposes using the following commands:

```
./config/configure.py --with-cc=gcc --with-fc=g77 --download-f-blas-lapack=1
--with-mpi=0
```

```
./config/configure.py --with-cc=gcc --with-fc=g77 --download-f-blas-lapack=1
--with-mpi=0 --with-debugging=0 COPTFLAGS="-O3"
```

For Peano we had to write our own makefile based on a sample makefile for PETSc since some libraries differed from those used under native linux. We included our makefile on the enclosed CD, so we will abstain from a library listing at this point.

2.2 Using PETSc

PETSc itself is fairly easy to handle: Basically, all it needs is a pointer to a method that evaluates a mathematical function $f : \mathbb{R}^n \to \mathbb{R}^n$ at a point $z \in \mathbb{R}^n$. Additionally, one may pass a pointer to another method evaluating the Jacobian function $J : \mathbb{R}^n \to \mathbb{R}^{n \times n}$ of f in $z \in \mathbb{R}^n$ - if no such function is given, PETSc offers a set of methods for calculating the Jacobian using finite differences.
Given these informations and provided an initial value $x_0 \in \mathbb{R}^n$, PETSc tries to find a root $x \in \mathbb{R}^n$ of the function f using a Newton technique. Therefore, in the n-th iteration a point $x_n \in \mathbb{R}^n$ is determined, which should be a closer approximation to x than x_{n-1}. For that purpose, $J(x_n)$ must be calculated and f - possibly multiple times - evaluated in each iteration.

2.3 Testing

Amongst others we tested the PETSc functionality with the following 2-dimensional function:

$$f : \begin{cases} \mathbb{R}^2 & \to & \mathbb{R}^2 \\ \begin{pmatrix} x \\ y \end{pmatrix} & \mapsto & \begin{pmatrix} x^2 - 4y + 7 \\ 3x^2 + \log(x) - 2 \end{pmatrix} \end{cases} \tag{1}$$

PETSc offers two different non-linear solvers: the trust region and the line search technique, although the implementation of trust regions is still in an ”experimental” state.

Additionally, it turned out to be much slower than line search and was therefore ignored by us for the rest of the project.
Line search is provided with four different variants: ”cubic”, ”quadratic”, ”basic” and ”basic with no norms”. As shown in table 1, all line search techniques converge in approximately the same time, though the ”basic with no norms” variant did not deliver any viable results in any of our tests. Thus we chose to use the cubic variant which is the default PETSc method as it is considered the most stable one.

	trust region	line search - cubic	line search - quadratic	line search - basic
time[s]	0.32363	0.0069520	0.0091453	0.0062743

Table 1: Convergence times

The non-linear solver of PETSc uses an internal linear solver which can also be configured, but as stability and performance cannot be judged appropriately for low dimensional equations like our test equation, we will postpone this topic to section 4.3 and appendix A.1, where the test results of the Navier-Stokes equation are discussed.

3 Wrapping the PETSc Methods

As mentioned in the introduction, Peano uses a finite element method to approximate the solution of the Navier Stokes equations. Therefore, a discrete grid is needed, which can be either adaptive or regular.
By now only regular grids are supported by the components of this project, so the adaptive components are ignored in the following explanations though Peano offers support for both grid types.

3.1 The Navier-Stokes Equations

The continuous, incompressible Navier-Stokes equations are defined by

$$\nabla \cdot \mathbf{u} = 0 \tag{2}$$

$$(\mathbf{u} \cdot \nabla) \cdot \mathbf{u} - \frac{1}{Re} \cdot \Delta \mathbf{u} + \nabla p = 0 \tag{3}$$

Those equations are discretized, resulting in the following equation:

$$B(\mathbf{u}, \mathbf{p}) := \begin{pmatrix} M \cdot \mathbf{u} \\ C(\mathbf{u}) \cdot \mathbf{u} + D \cdot \mathbf{u} - M^T \cdot \mathbf{p} \end{pmatrix} = 0 \tag{4}$$

where

- $B : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$ is the discrete Navier-Stokes function, with m being the grid dimension times the number of vertices and n the number of cells in the grid
- $\mathbf{u}$ is a vector containing the x and y components of the fluid velocity in every grid vertex
- $\mathbf{p}$ is a vector containing the pressure value in every cell
- M is an $n \times m$ matrix
- D is an $n \times n$ matrix
- C is a linear function from $\mathbb{R}^n$ to $\mathbb{R}^{n \times n}$

Additionally, we define $J : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^{(m+n)\times(m+n)}$ as the Jacobian of B.

3.2 Peano

The whole Peano implementation is object oriented, therefore all necessary methods have to be wrapped in classes. Figure 1 shows a simplified layout of the fluid component and the nonlinear solver component.

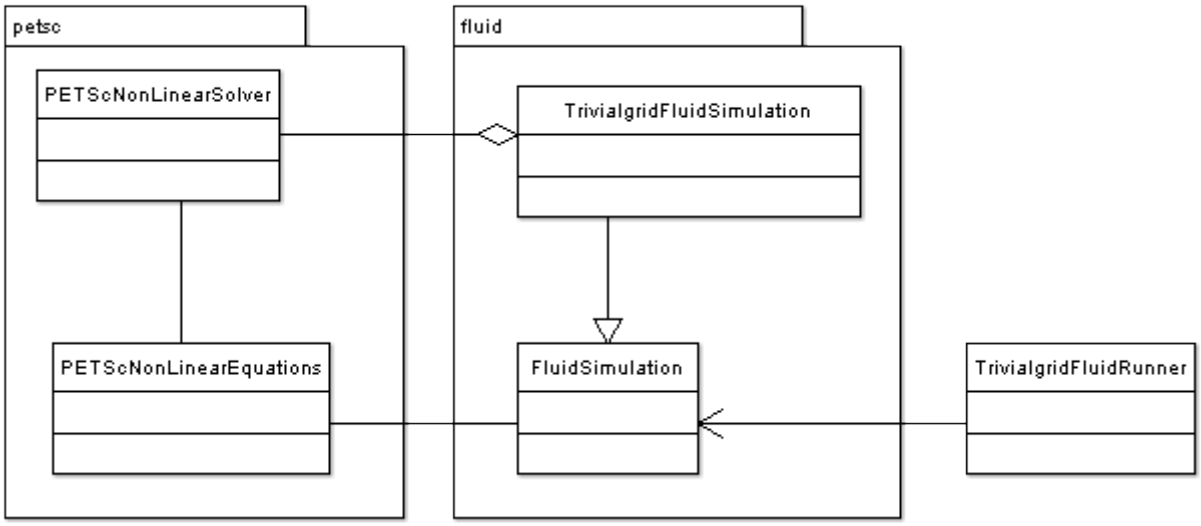

Figure 1: Simplified Peano class diagram

PETScNonLinearEquations essentially contains the methods evaluating B and J, function() and jacobianGrid(), which are used by the PETScNonLinearSolver class. The PETScNonLinearSolver class itself is independent of the fluid component - merely a pointer to the FluidSimulation class is passed through to function() and jacobianGrid(). FluidSimulation contains wrapping methods for access to the grid, whereas their functionality is provided by TrivialgridFluidSimulation for regular grids.
TrivialgridFluidRunner triggers the calculation in FluidSimulation, deciding whether a time step based or a steady state simulation will be executed.

3.3 The PETSc Nonlinear Solver

The PETScNonLinearSolver class provides wrapper methods for PETSc that can be used to solve equation systems without detailed knowledge about the PETSc API. The Usage of this class is explained in the doxyS documentation, so we'll focus on the mathematical details at this point.

When the solver is started, it performs the following actions, described in pseudocode:

Listing 1: PETScNonLinearSolver functionality

```
x := initial value
linear_solver := "GMRES" or "LSQR" or "GMRES with SPAI"
FOR i = 1 TO maximum number of nonlinear iterations DO
        Setup a LSE J(x) * d = b(x) using line search
        Solve LSE using linear_solver
        Calculate step width lambda
        x := x + lambda * d

        IF tolerance conditions are met THEN EXIT FOR
```

10 END FOR

where b(x) is a vector determined by line search.

Note: GMRES with SPAI is not yet supported.

3.4 Data Transfer

As PETSc stores its information in vectors, these data have to be passed back and forth from and to the Peano grid. In every iteration, the current values for **u** and **p** are stored into an array and uploaded to the grid, where B and J are computed using the existing peano grid functionality. Once the calculation is complete, B is read from the grid, stored into an array and handed back to PETSc as a result of the Navier Stokes equations, whereas J is stored into a matrix and made available to PETSc.

3.5 Peano Adapters

Peano uses adapter classes to guarantee the compatibility to different types of grids, such as regular or adaptive grids. Every adapter offers three methods for browsing the grid: One being called when a vertex is touched the first time, one being called when a vertex is touched the last time and one for the handling of each cell. For more complex operations, some of these methods use an adaptee for those calculations, as shown with CalculateJacobian and CalculateDivergence in Figure 2.
In the following, we will give a brief overview of the functionality of those classes:

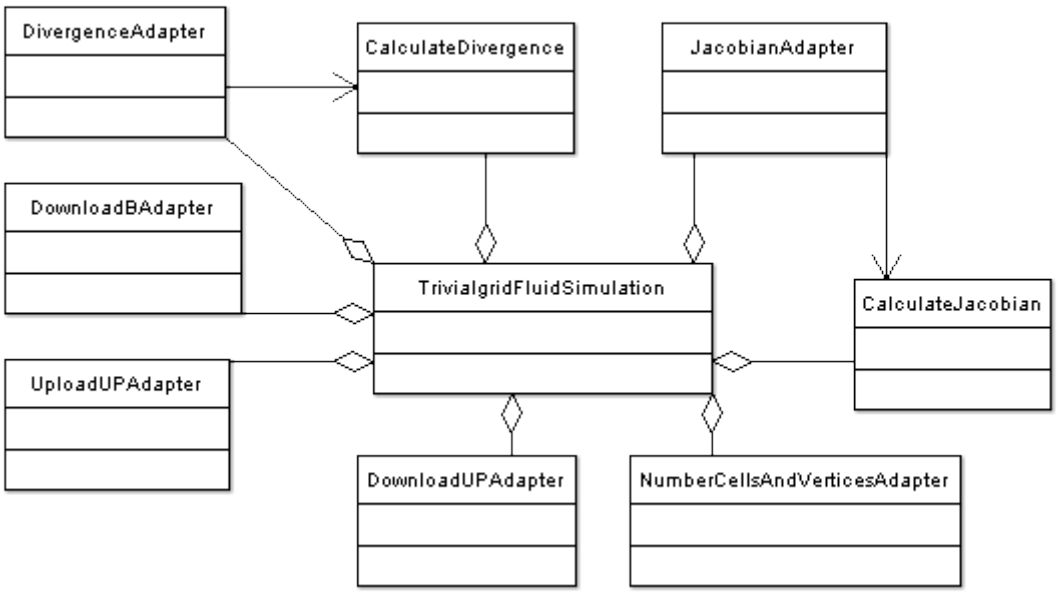

Figure 2: Adapter classes added in the context of this project

- **NumberCellsAndVerticesAdapter**
  Numbers all cells and vertices consecutively to define their position in the array.

- **DownloadUPAdapter**
  Reads current **u** and **p** from the grid to create an initial value for PETSc.

- **UploadUPAdapter**
  Copies the values of **u** and **p** from an array into the grid. This Adapter is called before the calculation begins.

- **DivergenceAdapter**
  Uses the CalculateDivergence class to calculate the divergence of every cell. The divergence in a point is defined by $\nabla \cdot \mathbf{u}$. In discretized form, this means it's calculated for a cell by multiplying an element matrix with the velocities of the vertices surrounding the cell and adding up the results.

- **DownloadBAdapter**
  Downloads the value of B from the grid into an array. This Adapter is called after the calculation of B has finished.

- **JacobianAdapter**
  Calculates the Jacobian matrix. This is described in detail in section 4.2.

Note: For the trivial grid, all adapters have the prefix "TrivialgridEventHandle2" attached to their names. As adaptive grids are not supported by now, only the trivial grid event handle adapters exist.

4 Optimization

As our tests have shown at this point, the implementation was much too slow and partially unstable by now, so several optimizations to improve performance and stability were necessary.

4.1 Adding Time Steps

The first step to improve the stability was tuning the initial value. In spite of generally using 0, Peano performs a number of time steps given by the configuration. The resulting values for $\mathbf{u}$ and $\mathbf{p}$ are used as an initial value for the calculation of the steady state afterwards.
After this change the solver worked more stable, as intended, however we could not observe any speed improvements.

4.2 Tuning the Jacobian Matrix Calculation

PETSc's built-in finite differences calculation turned out to be slow as it does not have any knowledge about the underlying grid of the scenario. Therefore it does not know which components in the array $(\mathbf{u}, \mathbf{p})$ contribute to one component of $B(\mathbf{u}, \mathbf{p})$ and assumes that all entries do when in fact only a small number does, which is independent of the grid size (≤ 22 values per component for 2D).

It is possible in PETSc to color the matrix columns for a more efficient calculation, but this optimization works only to a certain degree. There is still a lot of obsolete calculation which can be avoided.

The final matrix has the following layout:

$$J = \begin{pmatrix} \frac{\partial imp_1}{\partial \mathbf{u}_1} & \frac{\partial imp_1}{\partial \mathbf{u}_2} & \frac{\partial imp_1}{\partial \mathbf{p}} \\ \frac{\partial imp_2}{\partial \mathbf{u}_1} & \frac{\partial imp_2}{\partial \mathbf{u}_2} & \frac{\partial imp_2}{\partial \mathbf{p}} \\ \frac{\partial div}{\partial \mathbf{u}_1} & \frac{\partial div}{\partial \mathbf{u}_2} & \frac{\partial div}{\partial \mathbf{p}} \end{pmatrix}$$

where:

- $\mathbf{u}_1$, $\mathbf{u}_2$ are x, y components of $\mathbf{u}$
- $\mathbf{p}$ = pressure values
- imp_1, imp_2 are x, y components of the discrete impulse equation left hand side (see chapter 3.1, second component of $B(\mathbf{u}, \mathbf{p})$ in equation (4))
- div = divergence values

Note that $\frac{\partial div}{\partial \mathbf{p}} = 0$ for the whole block, as the divergence is independent of the pressure. We considered two simple options how to arrange $\mathbf{u}$ and $\mathbf{p}$ in the input array: either as $(\mathbf{u}, \mathbf{p})$ (variant A) or $(\mathbf{p}, \mathbf{u})$ (variant B), of which variant A proved to be more stable than B. Though it might seem odd that we deliberately chose to set a large number of diagonal entries to 0, the advantage of variant A lies in all other diagonal entries being guaranteed non-zeros. Variant B has neither the advantage nor the disadvantage of variant A, but it has a higher probability of zero diagonal entries.

As mentioned before, it is only necessary to compute the influence on each component of $B(\mathbf{u}, \mathbf{p})$ for a limited number of components of $(\mathbf{u}, \mathbf{p})$. For div in a cell these are $\mathbf{u}_1$ and $\mathbf{u}_2$ of the vertices adjacent to the cell (≤ 8 for 2D). For imp_1 and imp_2 in a vertex v these are all velocities of the vertices that share a cell with v ($\leq 9 \cdot 2 = 18$ for 2D) and the pressure values of the surrounding cells (≤ 4 for 2D).

We use finite differences for the computation, so for a function f and scalar x, $\frac{\partial f}{\partial x}$ would be calculated as $\frac{f(x+h)-f(x)}{h}$. The default value for h is 10^{-7} if $||x|| < 1$, else $||x|| \cdot 10^{-7}$.

So at first, we calculate $\frac{\partial div}{\partial \mathbf{u}_1}$ and $\frac{\partial div}{\partial \mathbf{u}_2}$ where div is calculated on a cell c and $\mathbf{u}_1$ and $\mathbf{u}_2$ are changed on all the vertices surrounding the cell c.

Next, we calculate $\frac{\partial imp_1}{\partial \mathbf{p}}$ and $\frac{\partial imp_2}{\partial \mathbf{p}}$ where $\mathbf{p}$ is changed on the cell c and imp_1 and imp_2 are calculated on all the vertices surrounding the cell c.

Finally, for every pair of vertices (v, w) that share the cell c, we calculate $\frac{\partial imp_1}{\partial \mathbf{u}_1}$, $\frac{\partial imp_1}{\partial \mathbf{u}_2}$, $\frac{\partial imp_2}{\partial \mathbf{u}_1}$ and $\frac{\partial imp_2}{\partial \mathbf{u}_2}$ where $\mathbf{u}_1$ and $\mathbf{u}_2$ are changed on v and imp_1 and imp_2 are calculated on w. The resulting values are added to the Jacobian entries.
'Added', because all vertices surrounding all cells adjacent to a vertex w contribute to the derivation in w, not just those of one cell. So the added contributions of each cell result in the derivation.

The result was a speed increase of the Jacobian calculation of up to factor 5 in comparison to the standard method, which confirmed our assumptions.

4.3 Configuring the Linear Solver

As mentioned in section 2.3, the PETSc non linear solver uses an internal linear solver which can be extracted and configured. Initially a least squares linear solver was used - this method solves the system $A^T \cdot A \cdot x = A^T \cdot b$. Its advantage lies in its stability, as $A^T \cdot A$ is guaranteed not to have zeros in the diagonal if the rank of A is full. However this method usually converges slowly, as $\kappa(A^T \cdot A) = \kappa(A)^2$.
We added the possibility to use GMRES as a linear solver instead. This method solves the system $A \cdot P^{-1} \cdot P \cdot x = b$ using right side ILU(dt) preconditioning. ILU(dt) is a variant of the ILU preconditioning method which drops values beneath a certain tolerance for improved performance and less memory requirement. As demonstrated in appendix A.1, this solver offers a better performance than LSQR for most problems and should therefore

be used if possible.

Here, we were able to increase the overall computation speed up to a factor of 4.

5 Conclusion And Prospects

In conclusion, one can say that all set goals have been achieved satisfactorily. Most test cases were solved with adequate stability, precision and speed, although there are still a few stability issues that could be addressed.
By now, the calculation of the steady state is limited to 2D regular grids, the solver is already prepared to handle adaptive and / or higher dimensional grids. The CalculateJacobian adapter would have to be extended to support higher dimensions, whereas it should require no changes for the adaptive grid. CalulateDivergence, PETScNonLinearEquations and the trivialgrid adapters would also need to be extended, but this should be a minor effort.
Parts of the functionality provided by TrivialgridFluidSimulation could also be moved to its super class FluidSimulation to support both grid types.

A Test Cases

All test cases were run on a AMD Athlon 64 3500+ (2,20 GHz) with 2 GB RAM under Windows XP SP2, using a release build of both Peano and PETSc. We chose three different scenarios to present: The free channel, a channel containing a cylinder-shaped obstacle and the driven cavity scenario.

A.1 Free Channel

This scenario was chosen mostly to show the difference between the GMRES and LSQR linear solver. As you can see, using GMRES results in a much faster calculation for this example. Also an analytical solution exists for this scenario, so we had the possibility of validating the correctness of our computation.

Configuration: resolution 220 x 41, Re = 20, relative tolerance = 10^{-7}

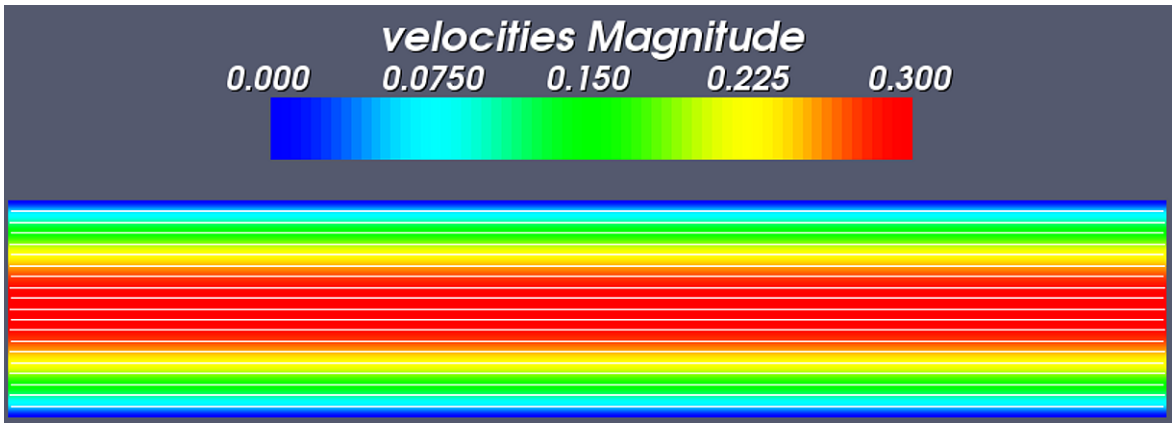

Figure 3: Free channel steady state

linear solver	runtime	number of iterations
GMRES	61 s	3
LSQR	283 s	19

Table 2: Test results free channel

A.2 Cylinder Obstacle

For this scenario, a reference value for the forces that affect the cylinder obstacle exists (see [2]). Thus it offers an excellent opportunity to check the precision of the computed solution, as also small differences which do not show up in other tests are noticeable.

Configuration: resolution 440 x 82 and 220 x 41, Re = 20, relative tolerance = 10^{-7}

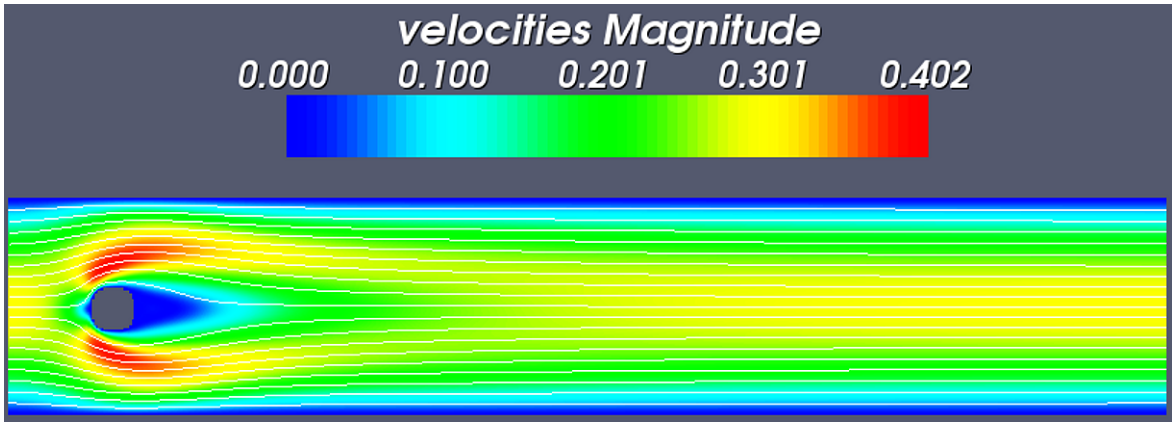

Figure 4: Cylinder obstacle, resolution 220 x 41

resolution	runtime	number of iterations	force vector x	force vector y
220 x 41	73 s	4	0.00991225	$4.21583 \cdot 10^{-05}$
440 x 82	760 s	4	0.00941885	$2.21526 \cdot 10^{-05}$
Force reference			0.01116	$2.14 \cdot 10^{-05}$

Table 3: Test results cylinder obstacle

A.3 Driven Cavity

Although this scenario does not provide any further information, we included it just to verify the functionality of our solver through an additional test.

Configuration: resolution 81 x 81, Re = 1, relative tolerance = 10^{-7}

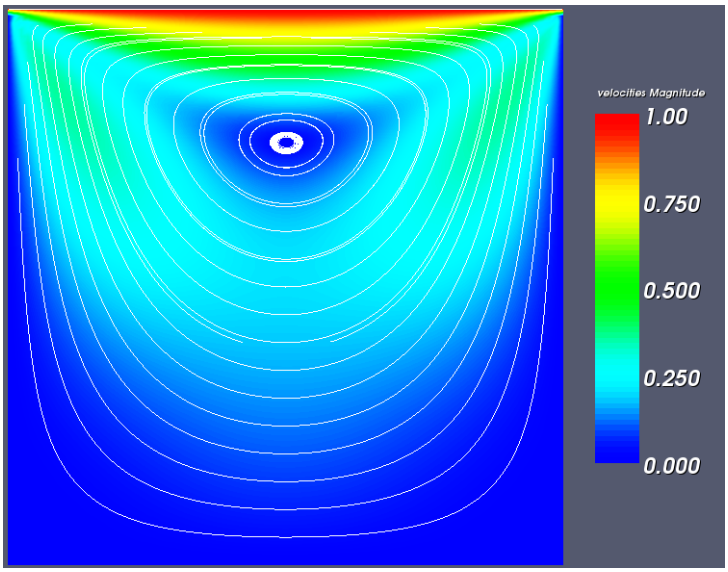

Figure 5: Driven cavity steady state

test case	runtime	number of iterations
Driven cavity	39 s	2

Table 4: Test results driven cavity

References

[1] http://www-unix.mcs.anl.gov/petsc/petsc-as/.

[2] M. Schäfer and S. Turek. *Flow Simulation with High-Performance Computers II*, volume 52 of *Notes on Numerical Fluid Mechanics*, chapter Benchmark Computations of Laminar Flow Around a Cylinder, pages 547–566. Vieweg, 1996.