Project Work

Time-implicit ODE-Solver
for the incompressible
Navier-Stokes Equations in Peano

Editor: Benjamin Uekermann
Examiner: Prof. Dr. Hans-Joachim Bungartz
Adviser: Dipl. Tech.-Math. Tobias Neckel
1 Introduction

To range my project work in a general context, I first of all explain in very few words, what Peano is about. At the Chair of Informatics V at the TU München, an adaptive grid (called Peano-Grid) for high-performances Partial Differential Equations (PDE) simulation was optimized over several years concerning the storage of data, the order the grid-cells are called and the way of refining the cell-structure. To this grid different applications were introduced. My project is placed in the field of fluid simulation. As a consequence of the optimized order of the grid cells, all calculations have to be performed locally on a single cell or vertex. To simplify and test such new techniques the Peano-Code also contains a regular cartesian grid (called Trivial-Grid). My project work uses strictly the Trivial-Grid. Nevertheless, all concrete implementation aspects may be reused or just have to be transferred to the corresponding Peano-Grid counterparts. At the beginning of my work two different ways of time dicretization were already possible: a semi-explicit Euler and a steady-state solution. The goal for me was to widen that zoo of Ordinary Differential Equation methods (ODE-methods) by implementing some implicit methods. They were necessary, because the steady-state method only allows to simulate time independent problems and by using the semi-explicit Euler you get a strict upper bord for the time step size to guarantee stablility.

1.1 Expectations

The main expectation was by making the discretising in space more and more finer, the semi-explicit Euler would lead to very small time steps while an implicit ODE method would allow bigger time steps. After the implementation some test runs should verify this assumption and should make a general statement for which simulation one should use what kind of ODE-method.

Another student project allocates an analytical way to compute the Jacobian matrix in the fluid simulation, but this has up to now little effect, because it only affects the steady-state solution. Implementing implicit ODE-methods could boost this effort, because they would also need a quite similar Jacobian matrix.

1.2 Structure of the Paper

The second chapter of this report concentrates on a theoretical view at implicit ODE-methods in the field of fluid simulation. As many parts for computation were similar to the already existing semi-explicit Euler and the steady-state solution, I will first explain these two methods. By introducing the implicit Θ-method, I will have a close look on which parts are really new and which parts could be used from the already existing code. Furthermore ways for calculating the time step size would be discussed, because implicit ODE-methods without an adaptive time step size would of course not be

1
better than explicit methods. The third chapter describes the implementation of these new contributions in Peano. There, expected as well as unexpected problems occurred. These problems and their solution will be presented. As my knowledge of code-design at the beginning of the project was very little, the implementation needed time as well as help. Therefore I want to thank Tobias Neckel for spending so much of his time to explain those things to me. Some test scenarios will be described in the fourth chapter with a close look on possible conclusions. Finally, I close the discussion with an outlook on further perspectives concerning peano.
2 Numerical Fundamentals

2.1 Discretizing Space

The starting point are the continuous Navier-Stokes-Equations for incompressible flow:

\[
\begin{align*}
\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \frac{1}{\nu} \Delta u + \nabla p &= 0 \\
\nabla \cdot u &= 0
\end{align*}
\]  

\[(\text{NSE})\]

The first (vector) equations are the momentum equations (that may be interpreted as a force balance) and the second (scalar) equation is the continuity equation (representing a mass balance). The goal is to find velocity \( u = u(x, t) \) and pressure \( p = p(x, t) \) functions, that satisfy both equations for all times \( t \in \mathbb{R} \) and space positions \( x \in \Omega \subset \mathbb{R}^2 \) or \( \mathbb{R}^3 \).

As this problem is not (yet) solvable analytically for most \( \Omega \), one is interested in approximate solutions that stem from some discretization in time and space (in order to reduce the problem to a finite-dimensional one). We discretize the space with a finite element approach for the velocities, while the pressure is treated as a Lagrange multiplier. The result is the set of semi-discrete Navier-Stokes-Equations: (see [Sim08] for details)

\[
\begin{align*}
A \cdot \dot{u}_h + C(u_h) \cdot u_h + D \cdot u_h + M^t \cdot p_h &= 0 \in \mathbb{R}^{D \cdot N} \\
&=: F(u_h) \\
M \cdot u_h &= 0 \in \mathbb{R}^Z
\end{align*}
\]  

\[(\text{DNSE})\]

\( D \) is the Dimension of the Scenario \((D = 2, 3)\), \( N \) is the number of vertices and \( Z \) is the Number of cells.

Remark:

- This set of equations is still time-dependent \((u = u(t), \ p = p(t))\).
- In all following equations the index \( h \), that emphasize the discrete space, in which the velocity and the pressure are living in, will be left out because of better reading arguments. This does not matter as everything from now on would treat the NSE problem as a space-discrete one.
- The matrices \( A, \ C, \ D, \ M \) are represented here, as global matrices applied to global vectors \( u, \ p \). As mentioned in the first Chapter the realisation in Peano is an element-wise one: these global matrices are never explicitly assembled, but local contributions are computed on every single cell or vertex. Thus, there is no need for storing global matrices, everything is stored in local attributes (such as \( F, \ \text{grad}P \)).
2.2 Discretizing Time

At the beginning of my project work, an explicit Euler method and a steady-state solution have been already implemented. As many calculation are similar for special methods, I could use a lot of things which were already implemented. To describe this, I now explain first of all how these two old methods work.

2.2.1 Semi-explicit Euler-Approach

The simplest way to discretize time is to use an explicit Euler. This does only work for the first equation, the second one has to be discretised in an implicit way, because the mass balance has to be satisfied for the new time step. This leads to the following equations: \( \tau \) meaning the actual time step size

\[
A \cdot \frac{u_{k+1} - u_k}{\tau} + C(u_k) \cdot u_k + D \cdot u_k + M^t \cdot p_k = 0 \\
M \cdot u_{k+1} = 0
\]

The advantage of this technique is that you can separate the calculation of the pressure and the velocities. Both equations of the DNSE has to be satisfied for the old values \( u_k \) and \( p_k \). Hence, by inserting the first equation into the time-derivative of the second equation you get the following relationship, often called Pressure-Poisson-Equation:

\[
M \cdot A^{-1} \cdot M^t \cdot p_k = M \cdot A^{-1} \left( -C(u_k) \cdot u_k - D \cdot u_k \right)
\]

So pressure values can be calculated from the velocity values by solving a linear equation system on every cell.

The second step then is to do a velocity update:

\[
A \cdot u_{k+1} = A \cdot u_k + \tau \cdot \left(-C(u_k \cdot u_k - D \cdot u_k) - M^t \cdot p_k \right)
\]

The disadvantage is that you have a strict upper border for the time step size \( \tau \) to guarantee stability as this is typical for explicit ODE methods. This is the reason why we will profit from implicit ODE-methods when we simulate the NSE on a very fine grid.

But before investigating the implicit Euler, have a look on the steady-state solution which was already implemented and which is quite similar to the implicit methods.
2.2.2 Steady-State Approach

If we assume that the solution of the NSE is time-independent after a certain time, the computation could be simplified. As we are only interested in the final solution, let \( \dot{u} = 0 \). So now the steady-state DNSE reads:

\[
B(x) := B\left(\begin{array}{c} u \\ p \end{array}\right) := \left(\begin{array}{c} C(u) \cdot u + D \cdot u + M^t \cdot p \\ M \cdot u \end{array}\right) = 0
\]

The vector \( B(x) \) is also denoted as the Navier-Stokes function. To solve this system of nonlinear equations the library PETSc is used([BBE+07]). PETSc uses a kind of Newton (-like) method to perform iterations starting from some initial guess \( x_0 \) to approximate the solution \( x \). So PETSc needs not only the function \( B(x) \), but also the Jacobi Matrix \( \nabla B \) and a good guess \( x_0 \) to start from. The Jacobian of \( B \) reads:

\[
\nabla B = \begin{pmatrix}
\frac{\partial F}{\partial u} & M^t \\
M & 0
\end{pmatrix}
\]

To compute this Jacobian there are at the moment three possibilities implemented: Finite Differences, Analytical Computation and Automatic Differentiation (via ADOLC see [WG05]). The differences of these three Methods in speed and accuracy would be discussed later.

In order to get a good initial guess, some so called „Pseudo-Time-Steps“, meaning explicit time steps with a small size, are calculated.
2.2.3 Implicit $\Theta$-Method

Let $\Theta \in [0; 1]$, the implicit $\Theta$-Method reads:

$$\tilde{B}(u_{k+1}, p_{k+1}) := \left( \frac{1}{\tau} A(u_{k+1} - u_k), 0 \right) + \left( \frac{\Theta(F(u_{k+1}) + \text{grad} P(p_{k+1})) + (1 - \Theta)(F(u_k) + \text{grad} P(p_k))}{M u_{k+1}} \right)$$

In praxis only certain $\Theta$ values are considered:

- $\Theta = 0$ semi-explicit Euler (FE) (see 2.2.1)
- $\Theta = 0.5$ implicit Trapezoidal Rule (TR)
- $\Theta = 1$ implicit Euler (BE)

BE and FE have order of convergence 1 in $u$ and $p$. TR has order 2 in $u$ and 1 in $p$ ([Sim08]). If $\Theta > 0$, this method is implicit, so we have to solve a nonlinear system of equations in every time step. Although the method looks quite complicated, the implementation can be simplified, because most parts exist already in the steady-state solution. For evaluating the function $\tilde{B}$, we only need to implement the first part $\frac{1}{\tau} A(u_{k+1} - u_k)$, because if we save $F(u_k)$ and $\text{grad} P(u_k)$ from the last time step on every vertex, we only have to compute $F(u_{k+1})$, $\text{grad} P(u_{k+1})$ and $M u_{k+1}$ in the second part, but this is similar to the steady-state solution.

For using again PETSc we further need the derivative of $\tilde{B}$. The Jacobian reads:

$$\nabla B = \frac{1}{\tau} \begin{pmatrix} A & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} \Theta \frac{\partial F}{\partial u} & \Theta M' \\ \frac{\partial F}{\partial M} & 0 \end{pmatrix}$$

Again, the second part is (quite) the same as in the steady-state solution. We only have to implement $\frac{1}{\tau} A$. That means, the whole implementation of the implicit $\Theta$-method could be reduced in the theoretic numerical view to the so-called the mass matrix $A$.

Now, in three insertions, some backround information will be given. First of all how to calculate the mass matrix $A$. Afterwards some numerical facts about BE and TR and finally some information how PETSc solves non-linear equation systems. It is necessary to consider this points while analysing numerical results (see 4).
As all computation is always performed locally on every cell, we only have to compute an element mass matrix for one cell. Let first of all the dimension of the scenario be 2 and the vertex indices surrounding one cell be called as follows:

If you look up the mass matrix $A$ in the deduction of the DNSE it reads:

$$A = \left( \langle \Phi_i, \Phi_j \rangle \right)_{0 \leq i,j \leq n}$$

with

$$\langle f(x), g(x) \rangle = \int_{R^2} f(x)g(x)dx$$

Considering only the 4 vertices surrounding one cell and using the local indices, the element mass matrix reads than:

$$A_{\text{elem}} = \left( \langle \Phi_i, \Phi_j \rangle \right)_{0 \leq i,j \leq 3}$$

Those $\Phi_i$ are the interpolation functions from the linearisation of the velocities:

$$u(x,t) = \sum_{i=1}^{N} \Phi_i(x)u_i(t)$$

with

$$u_i(t) = (u_0^i(t), u_1^i(t))^t$$

with the global vertex index $i$. Those functions have the form: (simple bilinear interpolation)

$$\Phi_i(x) = \begin{cases} \frac{x_0}{h_0} \frac{x_1}{h_1} & \text{for } x \in LU(i) \\ \frac{x_0}{h_0} \left( 1 - \frac{x_1}{h_1} \right) & \text{for } x \in RU(i) \\ \left( 1 - \frac{x_0}{h_0} \right) \left( 1 - \frac{x_1}{h_1} \right) & \text{for } x \in RA(i) \\ \frac{x_0}{h_0} (1 - \frac{x_1}{h_1}) & \text{for } x \in LA(i) \\ 0 & \text{else} \end{cases}$$

with sitting on the vertex $i$ the surrounding cells are called as follows:

Please note that the coordinates here are local coordinates of every cell where the origin is always on the vertex with the local number 0. But this does not make any trouble as we will see later.

So for the element mass matrix, we have to use the following computations:

(used the substitution $x_i = \frac{x_i}{h_i}$)

$$\int_0^1 x_i^2 dx_i = \frac{1}{3}$$

$$\int_0^1 x_i(1 - x_i)dx_i = \frac{1}{6}$$

$$\int_0^1 (1 - x_i)^2 dx_i = \frac{1}{3}$$

Finally we get for $u_0^i$:

(for all integrals the local coordinates fit perfectly together)

$$A'_{\text{elem}} = \frac{h_0 h_1}{36} \begin{pmatrix} 4 & 2 & 2 & 1 \\ 2 & 4 & 1 & 2 \\ 2 & 1 & 4 & 2 \\ 1 & 2 & 2 & 4 \end{pmatrix}$$

and for both components:

$$A_{\text{elem}} = \begin{pmatrix} A'_{\text{elem}} & 0 \\ 0 & A'_{\text{elem}} \end{pmatrix}$$

Now let us consider the case, that the dimension of the scenario is 3. This case does not differ much from the first case. Instead of bilinearisation we use trilinearisation. The final solution is:
\[
A'_{\text{elem}} = \frac{h_0 h_1 h_2}{216} \begin{pmatrix}
8 & 4 & 4 & 2 & 4 & 2 & 2 & 1 \\
4 & 8 & 2 & 4 & 2 & 4 & 1 & 2 \\
4 & 2 & 8 & 4 & 2 & 1 & 4 & 2 \\
2 & 4 & 4 & 8 & 1 & 2 & 2 & 4 \\
4 & 2 & 2 & 1 & 8 & 4 & 4 & 2 \\
2 & 4 & 1 & 2 & 4 & 8 & 2 & 4 \\
2 & 1 & 4 & 2 & 4 & 2 & 8 & 4 \\
1 & 2 & 2 & 4 & 2 & 4 & 4 & 8 
\end{pmatrix}
\]
and for all components:
\[
A_{\text{elem}} = \begin{pmatrix}
A'_{\text{elem}} & 0 & 0 \\
0 & A'_{\text{elem}} & 0 \\
0 & 0 & A'_{\text{elem}}
\end{pmatrix}
\]

**Insertion 2: Numerical Background of TR and BE**

In the following part general aspects of TR and BE are described. (see [Sim03])

For the general ODE
\[
d\frac{dx}{dt} y(x) = f(x, y(x))
\]
these two methods read:

\[
\begin{align*}
\text{(BE)} & \quad y_{k+1} = y_k + \tau f(x_{k+1}, y_{k+1}) \\
\text{(TR)} & \quad y_{k+1} = y_k + \frac{\tau}{2} (f(x_k, y_k) + f(x_{k+1}, y_{k+1}))
\end{align*}
\]

To analyse stability the following test equation is determined:
\[
d\frac{dx}{dt} y(x) = \lambda y(x), \quad Re(\lambda) \leq 0
\]

Most one-step methods like BE and TR applying on this ODE lead to the injection
\[
y_{k+1} = R(\tau \lambda) y_k
\]
R is called stability function. To get the analytic solution, R should be equal to the exponential function, but in truth R is only a padé-approximation of it. An ODE-method is called A-stable if it produces a monotonic decreasing sequence of solutions for every \( \tau \) for the test equation. Explicit methods like FE are not A-stable.

For BE we get:
\[
R(z)_{BE} = \frac{1}{1 - \frac{\tau \lambda}{z}}
\]

So BE is A-stable. Further \( R(\infty)_{BE} = 0 \), so this stability function approximates the analytic solution good for \( Re(\lambda) < 0 \), but bad for \( Re(\lambda) = 0 \) (please remark: \( Re() \) means in this context the real part of a complex number, not the Reynolds number). Hence, BE gives good numerical results for problems tending to a steady-state solution, while you can’t simulate oscillating problems for a long time interval, because everything would be damped to zero.

For TR we get:
\[
R(z)_{TR} = 1 + \frac{\tau}{2} \frac{1}{1 - \frac{\tau \lambda}{z}}
\]

Once again TR is A-stable, but by contrast to BE:
\[
|R(\infty)_{TR}| = 1.
\]
So good numerical results could be expected for oscillating problems, while TR is no good choice for problems tending to a steady-state solution.

Using this knowledge for our fluid-simulations, we simply can propose, as a first conclusion, to use TR for time dependent problems and BE for simulating the time evolution of converging scenarios (by contrast a steady-state simulation only calculates the final state). But we have to be careful with proclaiming simple rules, as the NSE is not a simple ODE. First of all we make errors by discretizing space. Secondly, the DNSE have an algebraic constraint - the mass balance. Gresho/Sani possibly assume bad stability for TR (see [GR00]) (page 798). [Sim08] (page 59) also cautions against too big time steps for TR as they could lead to numerical oscillating.
As mentioned before, we have to solve a nonlinear equation system for BE and TR in every timestep. To achieve this, the extern Library PETSc is used. PETSc was already used to solve the steady-state system, so on the one hand there were not much implementation changes necessary. For more details see [FR07]. On the other hand, by drawing the right conclusion out of numerical results it turned out to be interesting how PETSc works in the numerical point of view. In this insertion I will explain what kind of numerical method PETSc uses (compare [BBE07]). PETSc performs a Line Search Newton method with Cubic Backtracking. To understand what this means, let us consider a general nonlinear equation system:

\[ F(x) = 0 \]

Hence, the basic Newton method starting with an initial guess \( x_0 \) reads:

\[
i = 0, 1, \ldots \quad \text{Solve} \quad DF(x_i) \Delta x_i = -F(x_i) \quad \text{Update} \quad x_{i+1} = x_i + \Delta x_i
\]

This method has a quadratic order of convergence, but only converges locally. To get a global converging method we have to damp the update with a factor \( \omega \leq 1: x_{i+1} = x_i + \omega \Delta x_i \). The following figure should illustrate why \( \omega = 1 \) is not always the best choice.

The Newton direction \( \Delta x_i \) is an anticlimactic direction, e.g. \( \| F(x_i + \omega \Delta x_i) \|_2 < \| F(x_i) \|_2 \) for some (possibly small) \( \omega \). \( \omega = 1 \) is often a bad choice, particularly if \( x_0 \) is far away from the quadratic converging area. The best \( \omega \) is chosen by a Line Search. Let \( \omega_1, \omega_2, \ldots \) be a strictly monoton decreasing sequence. Increment \( \omega \), e.g. \( \omega = \omega_1, \omega_2, \ldots \) until:

\[
\| F(x_i + \omega \Delta x_i) \|_2 < (1 - \tau \omega) \| F(x_i) \|_2
\]

Remarks

- \( \tau \in (0, 1) \), the default value should be 0.5.
- To get a scaling-independent Line Search often \( \| DF(x_i)^{-1} F(x) \|_2 \) instead of \( \| F(x) \|_2 \) is used. (I think PETSc does not use this)
- Cubic Backtracking means that \( \omega_{i+1} = \tau^3 \omega_i \).
- If \( \omega \) gets too small, the calculation is stopped and the PETSc error \text{LSFailure} occurs.
- The nonlinear iteration stops successfully, if \( \| \Delta x_i \|_2 \leq \text{RTOL} \| x_i \|_2 + \text{ATOL} \)
- The Line Search Newton method is globally converging, but if there are more than one possible \( \hat{x} \), you cannot forecast to which \( \hat{x} \) the iteration converges.

To get the Newton direction \( \Delta x_i \), we have to solve a linear equation system. Here, PETSc again performs an iterative method. I will not explain the different possibilities PETSc offers here. [FR07] mentions GMRES (Generalized Minimal Residual Method) as the best choice.

It is important to have a clear overview over the different iterations BE and TR are performing:

For all further computation please remark that now a lower index always means the time iteration.
2.2.4 Calculating Adaptive Time Step Sizes

In the following methods to handle adaptive time step sizes are considered, first in detail for TR and then the similar technique for BE.

Trapezoidal Rule

The following variable-step implicit method is described by Gresho/Sani \cite{GR00}(pages 797-805). There a 2-step Adams-Bashfort-method (AB) is used as a predictor to calculate an adaptive time-step size for the Trapezoidal Rule. The predictor and the error control are only based on the velocity, the pressure is treated as an algebraic constraint.

For a general ODE $\dot{y} = f(t, y)$ the 2-steps Adams-Bashfort method with variable time step sizes reads:

$$y_{n+1} = y_n + \frac{\Delta t_n}{2} \left( 2 + \frac{\Delta t_n}{\Delta t_{n-1}} \right) f(y_n) - \frac{\Delta t_n}{\Delta t_{n-1}} f(y_{n-1})$$

This method is explicit, stable and has order 2, but we should not be discouraged by the word „explicit“. AB2 is only used as a predictor - set up new every time step. Hence, this does not lead to a stability border for our time step size. The local truncation error (LTE) - defined as the residual in the ODE-formula when the exact solution is inserted - of AB2 can be computed by Taylor matching. \cite{GR00} (pages 264-269)

$$LTE_n(AB2) = y_{n+1}^{\text{pred}} - y(t_{n+1}) = -\frac{1}{12} \left( 2 + 3 \frac{\Delta t_{n-1}}{\Delta t_n} \right) \Delta t_n^3 y_n^{(3)} + O(\Delta t_n^4). \quad (1)$$

Otherwise the LTE of TR:

$$LTE_n(TR) = y_{n+1} - y(t_{n+1}) = \frac{1}{12} \Delta t_n^3 y_n^{(3)} + O(\Delta t_n^4). \quad (2)$$

By neglecting the higher order terms $O(\Delta t_n^4)$ combining (1) and (2) leads to:

$$LTE_n(TR) \equiv \frac{y_{n+1} - y_{n+1}^{\text{pred}}}{3(1 + \frac{\Delta t_{n-1}}{\Delta t_n})}.$$ 

So by calculating the difference between TR and AB2 the LTE of TR could be approximated.

Further, (2) leads to:

$$\frac{\|LTE_{n+1}(TR)\|}{\|LTE_{n}(TR)\|} \leq \left( \frac{\Delta t_{n+1}}{\Delta t_n} \right)^3.$$ \quad (3)

That means if we want $\|LTE_{n+1}(TR)\| \leq ERR$, we simply calculate:

$$\Delta t_{n+1} = \Delta t_n \left( \frac{ERR}{\|LTE_n(TR)\|} \right)^{\frac{1}{3}}.$$
For using this approximation in an adaptive time step ODE-method, the first step differs from the others, because AB2 is a 2-step method so we do not have all necessary information at the beginning. To start we use the simple semi-explicit Euler as the predictor. The following steps have to be computed starting with initial velocity values $u_0$.

**Start-Up**

1. Calculate the start time step size $\Delta t_0$ by some heuristic approach or start with a conservatively small one (the integrator will quickly increase it to a good value).
2. Solve the PPE for $p_0$ (see 2.2.1):
   \[ MA^{-1}M'p_0 = MA^{-1}(-C(u_0)u_0 - Du_0). \]
3. Calculate the velocity update $\dot{u}_0 = A^{-1}(-F - \text{grad} P)$
   and perform it: $u_1^{pred} = u_0 + \Delta t_0 \cdot \dot{u}_0$.
4. Solve the nonlinear TR system with $u_1^{pred}$ as a first guess for $u_1$.
5. Invert TR to get the required AB2 velocity data:
   \[ u_1 = u_0 + \frac{\Delta t_0}{2} (\dot{u}_0 + \dot{u}_1) \Rightarrow \dot{u}_1 = \frac{2}{\Delta t_0} (u_1 - u_0) - \dot{u}_0. \]
6. Use the first time step size again (to assure stability): $\Delta t_1 = \Delta t_0$.

**General Step** $n = 1, 2, \ldots$

1. Predict the velocity with AB2:
   \[ u_{n+1}^{pred} = u_n + \frac{\Delta t_n}{2} \left( 2 + \frac{\Delta t_n}{\Delta t_{n-1}} \right) \dot{u}_n - \frac{\Delta t_n}{\Delta t_{n-1}} \dot{u}_{n-1}. \]
2. Solve the nonlinear TR system with $u_{n+1}^{pred}$ as a first guess for $u_{n+1}$.
3. Invert TR to get the AB2 data for the next step:
   \[ \dot{u}_{n+1} = \frac{2}{\Delta t_n} (u_{n+1} - u_n) - \dot{u}_n. \]
4. Compute LTE-estimate based on the velocity:
   \[ LTE_n(TR) = \frac{u_{n+1} - u_{n+1}^{pred}}{3 \left( 1 + \frac{\Delta t_{n+1}}{\Delta t_n} \right)}. \]
5. Compute the (potential) next step size:

\[ \Delta t_{n+1} = \Delta t_n \left( \frac{\epsilon}{\|LTE_n(TR)\|} \right)^\frac{1}{3}. \]

(use a smooth norm like the discrete \( l^2 \)-norm)

After a general step you have to distinguish the following cases:

1. \( \frac{\Delta t_{n+1}}{\Delta t_n} > 1 \) accept the increase
2. \( 0.8 < \frac{\Delta t_{n+1}}{\Delta t_n} < 1 \) accept the solution, but do not change \( \Delta t_{n+1} \)
3. \( \frac{\Delta t_{n+1}}{\Delta t_n} < 0.8 \) reject the current solution and repeat the step using \( \Delta t_{n+1} \)
4. \( \frac{\Delta t_{n+1}}{\Delta t_n} \ll 1 \) stop integration and print error message

You should also stop the integration and print an error message when there are more than 2 steps reduction in 1 time step.
**Implicit Euler**

A similar approach could be used for BE. Instead of AB2 the simple FE is used in every time step as the predictor.

**Start-Up**

1. Calculate the start time step size $\Delta t_0$.
2. Solve PPE to get $p_0$.
3. Calculate the velocity update $\dot{u}_0 = A^{-1}(-F - \text{grad}P)$.

**General Step $n = 0, 1, 2, \ldots$**

1. Predict the velocity with FE:
   \[ u_{n+1}^{\text{pred}} = u_n + \Delta t_n \dot{u}_n. \]
2. Solve the nonlinear BE system with $u_{n+1}^{\text{pred}}$ as a first guess for $u_{n+1}$.
3. Invert BE to get the data for the next step:
   \[ \dot{u}_{n+1} = \frac{u_{n+1} - u_n}{\Delta t_n}. \]
4. Compute LTE-estimate based on the velocity:
   \[ LTE_n(BE) = \frac{u_{n+1} - u_{n+1}^{\text{pred}}}{2}. \]
5. Compute next (potential) step size:
   \[ \Delta t_{n+1} = \Delta t_n \left( \frac{\epsilon}{\|LTE_n(BE)\|} \right)^{\frac{1}{2}}. \]
3 Implementation Aspects

3.1 General Problems and their Solutions

3.1.1 Component Design Changes

As mentioned in the last chapter, most computation from the steady-state solution could be used for the implicit θ-method, but in the application performing an implicit ODE-method differs much from calculating a steady-state solution. So we wanted to separate this two possibilities in the code. At the beginning of my project work, the fluid and steady-state components had the following general design:

![Diagram of the fluid component design](image)

Figure 1: Original basic design of the fluid component
To be able to perform an implicit run without having the steady-state component installed, we had to put all necessary functions and adapters from the steady-state into the fluid component:

![Diagram of fluid component]

Figure 2: New basic design of the fluid component

This design should be seen as a work-around as it is not really satisfying. Many points have more a "historical" background than real arguments for them in the status quo, e.g. all steady-state components. In the last chapter some suggestions to simplify the design will be discussed.
3.1.2 New Vertex and Cell Types

As mentioned in chapter one the basic philosophy of Peano is to calculate everything locally on every cell of the grid and to store all data locally on cells and vertices. For performing the semi-explicit Euler and the steady-state solution amongst others the following data is stored: (compare 2.2.1)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>Dimension</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>the actual velocity vectorial</td>
<td>Vertex</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>the actual pressure scalar</td>
<td>Cell</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>the diffusion/convection value vectorial</td>
<td>Vertex</td>
<td></td>
</tr>
<tr>
<td>gradP</td>
<td>the pressure gradient value vectorial</td>
<td>Vertex</td>
<td></td>
</tr>
</tbody>
</table>

For performing BE or TR with the adaptive time step method of 2.2.4 we need several further attributes, e.g. a value to store the contribution of the mass matrix - we call this value mass matrix momentum. We decided to set up two complete new vertex types - one for BE and one for TR - because we do not want to store useless information. The new vertex types have the following additional attributes: (all vectorial)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
<th>TR</th>
<th>BE</th>
</tr>
</thead>
<tbody>
<tr>
<td>oldu</td>
<td>the old velocities</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>oldF</td>
<td>the sum of the old F and gradP value</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>MMM</td>
<td>the mass matrix momentum</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>actual derivation value</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oldderivation value</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>oldv</td>
<td>the velocity predictor value</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1: New Vertex Attributes for TR and BE

Furthermore we needed a new cell type, because we have to store $p_{old}$. Implementation details for the adaptive time step method will be discribed later (see 3.4). As we wanted to distinguish vertex types at runtime, this led to many problems. We wanted to avoid new template-methods, so we had to put all adapters and grid instances out of the FluidSimulation. In the former version all adapters needed for nonlinear computations (steady-state and time-implicit) were attributes of the NonLinearTrivialgridFluidSimulation. Now those adapters are instanced in the TrivialgridFluidRunner and their references are packed into a repository. This repository is an attribut of the NonLinearTrivialgridFluidSimulation. Furthermore, we had to duplicate all functions of all adapters (also the explicit ones as we want to perform some explicit Pseudo-Time-Stepping) to make them compliant for all vertex and cell types. This code duplication is a minor one as adapters in general do not hold much implementation on
their own but delegate computations to corresponding classes. Being able to distinguish vertex-types at runtime has a big advantage. All real computations could be swapped out from the adapters into vertex functions. We could use the same adapters for all ODE-methods. That means the adapter are what they should be: „stupid“, but clearly arranged.

3.2 Enlarging of the Main Computation Functions

In chapter two it was figured out, that the main change between the steady-state solution and the implicit Θ-method is the mass matrix contribution in the Navier-Stokes-Function as well as in the Navier-Stokes-Jacobian. So we enlarged the implementation of these two function.

3.2.1 Calculation of the Navier-Stokes-Function

For computing the new $\tilde{B}(x)$ a new adapter called CalculateB4CrankNicholson was introduced. This adapter computes the F-values and the gradP-values just as the steady-state adapter CalculateB does, but further it computes the MMM-values by instancing the class CalculatingMMMEulerImplicit and calling the function accumulateMMMValues.

```cpp
template <class Vertex>
void fluid::CalculateMMMEulerImplicit::accumulateMMMValues(
    Vertex* vertex[NUMBER_OF_VERTICES_PER_ELEMENT],
    Vector h
){
    double hFactor = 1.0;
    for(int i=0; i<DIMENSIONS; i++){
        hFactor *= h(i);
    }

    Vector tmpMMM(0.0);
    for(int j=0; j<NUMBER_OF_VERTICES_PER_ELEMENT; j++){
        tmpMMM = Vector(0.0);
        for(int k=0; k<NUMBER_OF_VERTICES_PER_ELEMENT; k++){
            tmpMMM += ElementMassMatrix[j][k]
                * (vertex[k]->getU() - vertex[k]->getUOld())
                * (hFactor/_tau);
        }
        vertex[j]->addToMMM(tmpMMM);
    }
}
```

17
3.2.2 Calculation of the Navier-Stokes-Jacobian

For calculating the new Jacobian matrix we only implemented the new function `addElementMassMatrix()`, that adds the mass matrix part to the already existing steady-state Jacobian. We had to be careful as in the steady-state Jacobian the entries have a different order compared to the mass matrix. The steady-state Jacobian looks for example for 2D as follows:

\[
\begin{pmatrix}
\frac{\partial F^0_0}{\partial u} & \frac{\partial F^0_0}{\partial x} & \ldots & \frac{\partial F^0_0}{\partial a} & \frac{\partial F^0_0}{\partial p} \\
\frac{\partial F^1_0}{\partial u} & \frac{\partial F^1_0}{\partial x} & \ldots & \frac{\partial F^1_0}{\partial a} & \frac{\partial F^1_0}{\partial p} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
\frac{\partial F^0_1}{\partial u} & \frac{\partial F^0_1}{\partial x} & \ldots & \frac{\partial F^0_1}{\partial a} & \frac{\partial F^0_1}{\partial p} \\
\frac{\partial \text{div}}{\partial u} & \frac{\partial \text{div}}{\partial x} & \ldots & \frac{\partial \text{div}}{\partial a} & 0
\end{pmatrix}
\]

Hence, we had to add the mass matrix entries in a changed order.

```cpp
void fluid::CalculateJacobian::addElementMassMatrix(const Vector& h) {
    double hFactor = 1.0;
    for(int i=0; i<DIMENSIONS; i++) {
        hFactor *= h(i);
    }

    for (int j=0; j<NUMBER_OF_VERTICES_PER_ELEMENT; j++) {
        for(int d=0; d<DIMENSIONS; d++){
            _elementJacobian[(DIMENSIONS*i+d)*_dimensionOfElementJacobian + DIMENSIONS*j + d] += ElementMassMatrix[i][j] * (hFactor/_tau);
        }
    }
}
```
3.3 Overview of an Implicit Run

3.3.1 Time-Loop: The ODE Component

As mentioned before, at the beginning of my project work, the only implemented ODE-method was the semi-explicit Euler. By enlarging the zoo of ODE-methods, it seemed nicer to separate ODE-things from the Fluid component. A new component called ODE was introduced. A bridge pattern has been selected as the design of this component to make easy combinations of every implemented ODE-method with every application possible.

![Diagram of Bridge Pattern design of the ODE component]

The philosophy now is to put general ODE-things, which are independent from the application, into the ODE component. Specific ODE-things, which differ in every application as well as in every ODE-method, should be implemented on the lowest level - the vertices and cells, where we can, as discribed before, distinguish different ODE-methods. The runner classes of the fluid, like the main class `FluidSimulation` (compare Figure 2), should only hold organising things.

The main function of the `ODESolver` is `solve()` (see also Figure 6). This function calls some preparing steps as `setInitialValue()` and then triggers the time-loop. In every iterations it calls amongst others `calculateTimeStepSize()` and `solveOneTimeStep()`. This last function now calls `implementDerivativeAndUpdate()` in a different way for every ODE-method. All concrete contents are outreached via the bridge pattern to the applications.
3.3.2 Performing One Implicit Time-Step: The Fluid Component

The following figures should give a basic overview what happens, when one implicit time step is performed. This should make it possible for other students who are beginning to work on the fluid component to get a first impression before looking at the code. For me such figures were very useful. The main function to explain is `implementDerivativeAndUpdate()`: (see Figure 4)

![Diagram](image)

Figure 4: General Overview: `solveOneTimestep`
To use PETSc as an extern library to solve nonlinear systems like $f(x) = 0$, we need to set two function references in PETSc: one to evaluate $f(x)$ and one to compute the derivative $\nabla f$. For more details on that see the IDP work about the steady-state solution([FR07]). These two functions are in this context `calculateNavierStokesFunction()` and `CalculateNavierStokesJacobian()`. Figure 5 gives an overview over one PETSc Iteration.

![Figure 5: Overview: One PETSc-Iteration](image)

### 3.4 Calculation of an Adaptive Time Step Size

In this section I will first explain the general order of operations which are performed when we use an implicit ODE-method with an adaptive time step size. After that I will explain the concrete actions for the TR method of 2.2.4.

We need two adapters, one to run over the grid before the nonlinear TR system is solved and one afterwards (see also Figure 4). If we want to use any multi-step method we have
to calculate different first steps (e.g. AB2 as the predictor for TR). So I implemented a function called `calculateStartSteps()`. We want to use the same adapters for all steps and pass the decision which step we want to calculate to the vertex level. Hence, I introduced a static vertex attribut named `stepCase`. The value of this integer decides which step is performed. (see Table 2)

<table>
<thead>
<tr>
<th>Value</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>−2</td>
<td>Explicit Update</td>
</tr>
<tr>
<td>−1</td>
<td>Reloading Step (after rejection)</td>
</tr>
<tr>
<td>0</td>
<td>General Implicit Step</td>
</tr>
<tr>
<td>1, 2, ...</td>
<td>Implicit Start Steps</td>
</tr>
</tbody>
</table>

Table 2: Explanation of concrete values of the static vertex attribut `stepCase`

To get a better general overview look at Figure 6. Here the whole time-loop of the ODE component is compared to the chain of function calls in the fluid component. In addition to 4 the adaptive time step elements are also illustrated.
Now let us concentrate on handling the concrete actions for the adaptiv TR method. The points of 2.2.4 are splitted up into three functions: The vertex function `handleODEInitialise()`, which is called in the Initialise Adapter, the vertex function `handleODEUpdate()`, which is called in the Update Adapter and the `NonLinearTGFluidSimulation` function `evaluateImplicitTimestepSizeCriterion()`.

Figure 6: Overview: Implicit Methods with an Adaptive Time Step Size (TSS)
handleODEInitialise()

<table>
<thead>
<tr>
<th>_stepCase</th>
<th>operations</th>
</tr>
</thead>
</table>
| -1        | \[ \dot{u} = \dot{u}_{old} \]  
           | \[ \dot{u}_{old} = u_{pred} \]  
           | \[ u_{pred} = u_{old} + \frac{\Delta t}{2} (2 + \frac{\Delta t}{\Delta t_{old}} \cdot \dot{u} - \frac{\Delta t}{\Delta t_{old}} \cdot \dot{u}_{old}) \]  
           | \[ u = u_{pred} \]  |
| 0         | \[ f_{old} = F + \text{grad}P \]  
           | \[ u_{old} = u \]  
           | \[ u_{pred} = u_{old} + \frac{\Delta t}{2} (2 + \frac{\Delta t}{\Delta t_{old}} \cdot \dot{u} - \frac{\Delta t}{\Delta t_{old}} \cdot \dot{u}_{old}) \]  
           | \[ u = u_{pred} \]  |
| 1         | \[ \dot{u}_{old} = A^{-1}(-F - \text{grad}P) \]  
           | \[ u_{old} = u \]  
           | \[ f_{old} = F + \text{grad}P \]  
           | \[ u = u + \tau \cdot \dot{u}_{old} \]  |

handleODEUpdate()

<table>
<thead>
<tr>
<th>_stepCase</th>
<th>operations</th>
</tr>
</thead>
</table>
| -2        | \[ \dot{u}_{old} = A^{-1}(-F - \text{grad}P) \]  
           | \[ u = u + \tau \cdot \dot{u}_{old} \]  |
| -1        | \[ u_{tmp} = \dot{u}_{old} \]  
           | \[ \dot{u} = \frac{2}{\Delta t} (u - u_{old}) - \dot{u}_{old} \]  
           | \[ u_{pred} = \frac{u - u_{pred}}{3(1 + \frac{\Delta t}{\Delta t_{old}}) (u + u_{sec})} \]  
           | \[ CUM = u_{pred} \]  
           | \[ u_{pred} = u_{tmp} \]  |
| 0         | \[ \dot{u} = \frac{2}{\Delta t} (u - u_{old}) - \dot{u}_{old} \]  |
evaluateImplicitTSC()

\[
\Delta t = \Delta t_{\text{old}} \cdot \left( \frac{\epsilon}{\|u_{\text{pred}}\|_2} \right)^{\frac{1}{3}}
\]

**Remark**

- **Notation:** \( u_{\text{sec}} \) means a security value to avoid overflow. \( \frac{u}{v} \) of two vectors \( u, v \) should be treated as a division in each component. \( \text{CUM} \) should symbolize calculateUMeasurement. This function adds the value to the discrete \( l^2 \)-norm. \( u_{\text{tmp}} \) is a only a simple variable, no attribut.

- The attribut \( u_{\text{pred}} \) is misused at the end of \( \text{handleODEUpdate} \) in order to store the old derivative information (to reconstruct it in case of step rejection). This is no problem as the predictor value has no use after \( \text{CUM} \).

- In case of a reloading step after rejection \( (_{\text{stepCase}} = -1) \), also the pressure values have to be reconstructed \( (p = p_{\text{old}}) \). This happens in a cell function, again named \( \text{handleODEInitialise()} \), which is called in the Initialise Adapter.

- The adaptive time step method for BE (also described in 2.2.4) is not yet supported in the code. It should be no big deal to add it, but we decided no to do it at the moment. As described in 2.2.3 we expect little utilization for it.

- To optimize the performance, Gresho/Sani ([GR00]) suggest not to calculate a new Jacobian Matrix in every time step. The decision to calculate a new one should depend on the quality of the last time step. At the moment this happens in every time step, even in a reloading step.

- Here a relative norm was chosen for measurement. For the Benchmark Scenarios (see Chapter 4) an absolute norm brought better performance.
4 Numerical Results

In this part, numerical results of different scenarios are presented. This part was the most challenging one in the mathematical point of view, because the integrators BE and TR applied on the (DNSE) do not work as dependably as in a simple ODE context. I decided also to mention sometimes bad numerical results to show and discuss the problems that occured.

4.1 Testing Implicit Methods

The first aim was simply to test the correct implementation. For that reason I tested BE and TR on a very simple scenario: a free channel in 2D with the fluid streaming in from the left side with a parabolic velocity contribution. The characteristic of this scenario is that we can compute an analytic steady-state solution the simulation should tend to. The limit constellation should look as follows: From the left to the right side the parabolic velocity contribution should be transported one to one while the pressure should show a linear decay.

For simplification only a 9x9 grid is simulated. (see Table 3)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable Mode</td>
<td>Debug</td>
</tr>
<tr>
<td>Scenarion</td>
<td>Free Channel</td>
</tr>
<tr>
<td>Trivial-Grid Dimensions</td>
<td>9x9</td>
</tr>
<tr>
<td>Linear Solver</td>
<td>GMRES</td>
</tr>
<tr>
<td>Preconditioner</td>
<td>ILU</td>
</tr>
<tr>
<td>Jacobian Type</td>
<td>Finite Differences</td>
</tr>
<tr>
<td>Max Linear Iterations</td>
<td>1000</td>
</tr>
<tr>
<td>Max Nonlinear Iterations</td>
<td>1000</td>
</tr>
<tr>
<td>Relative Convergence Tol</td>
<td>1.0e-7</td>
</tr>
<tr>
<td>Absolute Convergence Tol</td>
<td>1.0e-8</td>
</tr>
<tr>
<td>Solution Change Tol</td>
<td>1.0e-8</td>
</tr>
<tr>
<td>Start Time</td>
<td>0.0</td>
</tr>
<tr>
<td>Time Steps</td>
<td>10</td>
</tr>
<tr>
<td>Tau</td>
<td>0.02</td>
</tr>
<tr>
<td>Pseudo-Time-Steps</td>
<td>0</td>
</tr>
<tr>
<td>Re</td>
<td>1.0</td>
</tr>
<tr>
<td>Velocity Mean Value</td>
<td>1.0</td>
</tr>
<tr>
<td>Velocity Profile</td>
<td>Parabola</td>
</tr>
<tr>
<td>Init Velocity Everywhere</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3: Configuration for testing correct implementation
Explanations  As most variables in the configuration are self-explanatory, I will only comment those which describe the initial values. The pressure is always initialized with zero in every cell. By contrast the velocity values can either be set only in the first line of vertices on the left side or in every vertex. The variable *Init Velocity Everywhere* holds this information. *Velocity Mean Value* then is simply the arithmetic mean of the initial velocities. The contribution of the norm of those velocity values is described by *Velocity Profile. Parabola* means that the contribution in every vertical line of vertices is a parabolic one.

Results  In Table 4 for both time integrators BE and TR the maximum and minimum pressure values for every time step are registered. The analytic steady-state solution would be $p_{\text{max}} = 11\frac{1}{3}$ and $p_{\text{min}} = 2\frac{2}{3}$ (see [Nec05]). So first you can see that BE quickly converges into the steady-state solution. By contrast TR shows - at first glance - a very strange behavior. The pressure values oscillate. We expected bad performance for TR on steady-state scenarios (see 2.2.3 Insertion 2), but in this range? More simulation runs showed that the better the initial value is, the smaller the oscillating range gets. Gresho/Sani ([GR00]) also mentions such a numerical feature for TR if the timestep-size is too big. But this does not include that smaller timestep-sizes do really succeed. In truth they often lead to instability in the nonlinear equation system. So as a first statement I can say that TR is totally inappropriate for steady-state scenarios.

<table>
<thead>
<tr>
<th></th>
<th>BE</th>
<th>TR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p_{\text{max}}$</td>
<td>$p_{\text{min}}$</td>
</tr>
<tr>
<td>1</td>
<td>67.1</td>
<td>0.000</td>
</tr>
<tr>
<td>2</td>
<td>12.4</td>
<td>0.739</td>
</tr>
<tr>
<td>3</td>
<td>11.7</td>
<td>0.690</td>
</tr>
<tr>
<td>4</td>
<td>11.4</td>
<td>0.675</td>
</tr>
<tr>
<td>5</td>
<td>11.4</td>
<td>0.670</td>
</tr>
<tr>
<td>6</td>
<td>11.3</td>
<td>0.668</td>
</tr>
<tr>
<td>7</td>
<td>11.3</td>
<td>0.667</td>
</tr>
<tr>
<td>8</td>
<td>11.3</td>
<td>0.667</td>
</tr>
<tr>
<td>9</td>
<td>11.3</td>
<td>0.667</td>
</tr>
<tr>
<td>10</td>
<td>11.3</td>
<td>0.667</td>
</tr>
</tbody>
</table>

Table 4: Results for testing correct implementation
Finally the correct implementation could be proved „almost surely“ (remaining scepticism on TR could be excluded by the following benchmark scenarios).

Figure 7: Free Channel Scenario, final BE solution
4.2 Computation of the Benchmark Scenarios

For getting detailed information how good BE and TR perform, the Benchmark Scenarios are calculated ([TS96]). Here we got a channel with an obstacle - a cylinder in 3D and a circle in 2D - on the left side. The specific of this scenario is that the position of the cylinder is not vertically centred. It is a little bit dislocated. Hence, there is a force working on the cylinder. We have good information (out of certain physic experiments and simulations with a very high resolution) about this force values. Furthermore, we can compute this values out of the pressure values around the cylinder. So we have a good indication for the accuracy of our simulation. There are two different Benchmark Scenarios. Firstly, one with a low Reynolds Number \( Re = 20 \). This scenario has a steady-state solution. Secondly, one with a higher Reynolds Number \( Re = 100 \). By contrast, this scenario has a time-dependent solution.

4.2.1 Trivial-Grid steady-state Benchmark Scenario

Look at Table 5 for the general configuration of the first Benchmark Scenario I computed.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable Mode</td>
<td>Release</td>
</tr>
<tr>
<td>Scenario</td>
<td>Obstacle in a Channel</td>
</tr>
<tr>
<td>Trivial-Grid Dimensions</td>
<td>220x41</td>
</tr>
<tr>
<td>Linear Solver</td>
<td>GMRES</td>
</tr>
<tr>
<td>Preconditioner</td>
<td>ILU</td>
</tr>
<tr>
<td>Jacobian Type</td>
<td>Finite Differences</td>
</tr>
<tr>
<td>Max Linear Iterations</td>
<td>10000</td>
</tr>
<tr>
<td>Max Nonlinear Iterations</td>
<td>100</td>
</tr>
<tr>
<td>Relative Convergence Tol</td>
<td>1.0e-7</td>
</tr>
<tr>
<td>Absolute Convergence Tol</td>
<td>1.0e-8</td>
</tr>
<tr>
<td>Solution Change Tol</td>
<td>1.0e-8</td>
</tr>
<tr>
<td>Start Time</td>
<td>0.0</td>
</tr>
<tr>
<td>End Time</td>
<td>10.0</td>
</tr>
<tr>
<td>Pseudo-Time-Steps</td>
<td>0</td>
</tr>
<tr>
<td>Re</td>
<td>20.0</td>
</tr>
<tr>
<td>Velocity Mean Value</td>
<td>0.2</td>
</tr>
<tr>
<td>Velocity Profile</td>
<td>Parabola</td>
</tr>
</tbody>
</table>

Table 5: Configuration of the first Benchmark Scenario

We have seen before that BE does a good job on steady-state scenarios. The remaining questions are: „How good?“ and „How does the accuracy depend on the time-step size?“. To determine this I used BE with different constant time step sizes. The velocity values were initialized everywhere. Table 6 lists the results. We can see that this results are quite independent from the time step size. Only \( \tau = 10.0 \) ends in different pressure and
force values. Furthermore, if we compare this BE results with FE results (they are listed in Table 7), we see that both time integrators lead to the same final state. Finally, we can proclaim that a huge time step size like $\tau = 1e8$ (which leads to the simple steady-state solver) is the best choice to simulate such a scenario which converges into a steady-state solution (if we are only interested in the final state).

<table>
<thead>
<tr>
<th>Simulation Time $\tau$</th>
<th>Time Steps</th>
<th>Run Time</th>
<th>Pressure Values $p_{\text{min}}$</th>
<th>Force Coefficients $cd$, $cl$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.02</td>
<td>500</td>
<td>33 min</td>
<td>-0.0322 0.122</td>
<td>4.96 0.0211</td>
</tr>
<tr>
<td>0.2</td>
<td>50</td>
<td>23 min</td>
<td>-0.0322 0.122</td>
<td>4.96 0.0211</td>
</tr>
<tr>
<td>2.0</td>
<td>5</td>
<td>55 sec</td>
<td>-0.0324 0.122</td>
<td>4.96 0.0211</td>
</tr>
<tr>
<td>10.0</td>
<td>1</td>
<td>24 sec</td>
<td>-0.0333 0.124</td>
<td>5.15 0.0262</td>
</tr>
<tr>
<td>1e8</td>
<td>1</td>
<td>25 sec</td>
<td>-0.0322 0.122</td>
<td>4.96 0.0211</td>
</tr>
</tbody>
</table>

Table 6: Benchmark Scenario (220x41 Re20): Comparison of different time step sizes for BE

Figure 8 shows the final image of the scenario with $\tau = 0.02$. There are also streamlines illustrated (created by some Runge-Kutta integrator).

In the second sequence of simulation runs I compared BE and FE on the basis of this scenario. Now I initialized the velocity values only on the left side. This poses a bigger challenge for the integrators. Hence, differences are more visible. The results are listed on Table 7. The time for one time step of BE and FE differs in an approximate factor 100, but we have seen before that there are scenarios where BE allows a very big time step size. So I can proclaim that BE is a good alternative for FE for calculating steady-state scenarios, foremost for fine space discretization. Here FE would need a very small time-step size to assure stability while BE allows bigger time-step sizes (compare the similar results for TR later).
Table 7: Benchmark Scenario (220x41 Re20): Comparision of BE and FE

Remark The reference force values are cd = 5.58 and cl = 0.0107. The difference to our simulated values are caused by the discretization in space. 220x41 is no really high resolution, but should only be a first impression. This discretization error also explains the negative pressure values.

For the last simulation run on this scenario I tried TR. We have seen that TR is no good choice for the simple 9x9 free channel, so it is clear that it will be no good choice here either. Nevertheless I registered the results on Table 8. It is important to get an overall impression how the typical TR oscillating looks like ([GR00] calles that effect „ringing“). As mentioned before TR needs a good initial value to controll the ringing range. To achieve that value I simulated 50 FE pseudo time-steps.

Table 8: Benchmark Scenario (220x41 Re20): Force Coefficients

<table>
<thead>
<tr>
<th>step</th>
<th>cd</th>
<th>cl</th>
</tr>
</thead>
<tbody>
<tr>
<td>491</td>
<td>4.9645</td>
<td>0.020587</td>
</tr>
<tr>
<td>492</td>
<td>4.9477</td>
<td>0.021572</td>
</tr>
<tr>
<td>493</td>
<td>4.9645</td>
<td>0.020588</td>
</tr>
<tr>
<td>494</td>
<td>4.9477</td>
<td>0.021571</td>
</tr>
<tr>
<td>495</td>
<td>4.9645</td>
<td>0.020589</td>
</tr>
<tr>
<td>496</td>
<td>4.9477</td>
<td>0.021570</td>
</tr>
<tr>
<td>497</td>
<td>4.9645</td>
<td>0.020589</td>
</tr>
<tr>
<td>498</td>
<td>4.9478</td>
<td>0.021569</td>
</tr>
<tr>
<td>499</td>
<td>4.9645</td>
<td>0.020590</td>
</tr>
<tr>
<td>500</td>
<td>4.9478</td>
<td>0.021568</td>
</tr>
</tbody>
</table>

Results You can clearly see the ringing in the force coefficients. Furthermore, there is still some converging after 500 time-steps, but very slow ($\leq 10^{-4}$ for the drag coefficient, $\leq 10^{-6}$ for the lift coefficient). More interesting is the oscillating range. It is $\approx 1.7 \cdot 10^{-2}$ for the drag coefficient and $\approx 1.0 \cdot 10^{-3}$ for the lift coefficient. So the relative ringing range of the lift coefficient is 10 times worse than the one of the drag coefficient.
4.2.2 Trivial-Grid time-dependent Benchmark Scenario

What we have seen up to now is that BE is a very robust integrator and can even handle very bad initial values. By contrast TR shows the ringing problem which strongly depends on the initial value. So should we give up TR and only use BE and FE? No, surely not. The following scenarios shows that BE is in a way „too robust“ to simulate time-dependent problems. The numerical damping always leads to a numerical steady-state solution, which could be interpreted as a certain mean value. But for most possible applications were not interested in that mean value - we are interested in the time evolution.

If we calculate the Benchmark Scenario from the last chapter with Reynoldsnumber 100 we get a time-dependent solution. To check the accuracy of our simulations we once again only consider the force coefficients on the cylinder. I compare the new BE and TR data with the already existing FE simulation runs, but you should not treat the FE data as „the right one“. If you simulate such a small scenario like this 220x41 with FE do not use the biggest possible time-sep size. The time discretization error would get to big. Look on Table 9 and 10 for the configuration details of the FE respectively BE run.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrator</td>
<td>FE</td>
</tr>
<tr>
<td>Trivial-Grid Dimensions</td>
<td>220x41</td>
</tr>
<tr>
<td>Start Time</td>
<td>0.0</td>
</tr>
<tr>
<td>End Time</td>
<td>10.0</td>
</tr>
<tr>
<td>Time Step Size</td>
<td>0.0002</td>
</tr>
<tr>
<td>Number Of Time Steps</td>
<td>50000</td>
</tr>
<tr>
<td>Re</td>
<td>100.0</td>
</tr>
</tbody>
</table>

Table 9: Configuration FE Benchmark Scenario (220x41 Re100)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrator</td>
<td>BE</td>
</tr>
<tr>
<td>Trivial-Grid Dimensions</td>
<td>220x41</td>
</tr>
<tr>
<td>Start Time</td>
<td>0.0</td>
</tr>
<tr>
<td>End Time</td>
<td>10.0</td>
</tr>
<tr>
<td>Time Step Size</td>
<td>0.02</td>
</tr>
<tr>
<td>Number Of Time Steps</td>
<td>500</td>
</tr>
<tr>
<td>Re</td>
<td>100.0</td>
</tr>
<tr>
<td>Number of Pseudo Time Steps</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 10: Configuration BE Benchmark Scenario (220x41 Re100)
Figure 9: Benchmark Scenario (220x41 Re100), FE: Drag Coefficient

Figure 10: Benchmark Scenario (220x41 Re100), FE: Lift Coefficient
Results  Please compare the lift coefficient plots of BE and FE. There, you can clearly see the numerical damping BE produces. The drag coefficient shows a similar behavior. So we know now that BE cannot handle time dependent problems.

How can TR handle with this scenario? We already know that TR needs a good initial values to start from. Look on Table 11 for the configuration details.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrator</td>
<td>TR</td>
</tr>
<tr>
<td>Trivial-Grid Dimensions</td>
<td>220x41</td>
</tr>
<tr>
<td>Start Time</td>
<td>0.0</td>
</tr>
<tr>
<td>End Time</td>
<td>10.0</td>
</tr>
<tr>
<td>Time Step Size</td>
<td>0.02</td>
</tr>
<tr>
<td>Number Of Time Steps</td>
<td>500</td>
</tr>
<tr>
<td>Re</td>
<td>100.0</td>
</tr>
<tr>
<td>Number of Pseudo Time Steps</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 11: Configuration TR Benchmark Scenario (220x41 Re100)
Figure 12: Benchmark Scenario (220x41 Re100), TR: Drag Coefficient

Figure 13: Benchmark Scenario (220x41 Re100), TR: Lift Coefficient
**Results**  Here we got interesting results. Firstly, the lift coefficient is quite the same. I think there is no use for spending too much time on discussing why the highest level produced be TR is a little bit lower than the FE one. That could be a result of the different time steps. Further, maybe, the TR values are better than the FE ones. Secondly, let us have a close look on the drag coefficient. The real oscillating is superposed by the ringing. That is not too good, but also not too bad. Anyway, it is much better than the BE results. Perhaps, it is possible to filter the ringing out of the plot. Why can we see some ringing in the drag coefficient but not in the lift coefficient? The answer is that the drag oscillation has a higher frequency and a lower amplitude than the lift oscillation. So there is also a ringing on the lift coefficient, but it is relatively just too small to see it. That is exact what we want. The next question is clear: „How can we change the TR run, so that the ringing in the drag coefficient also gets smaller?“

The first attempt was to use a smaller time-step size. To be sure I used a ten-time smaller one ($\tau = 0.002$). Look at the corresponding drag coefficient plot. Nothing changed.

![Figure 14: Benchmark Scenario (220x41 Re100), TR: Drag Coefficient, $\tau = 0.002$](image)

The second attempt was to produce a better initial value. Remember the 9x9 Free Channel Scenario: there this procedure helped. In principle, there are two ways of achieve this. Firstly to calculate enough Pseudo Time Steps so that we can bridge the complete starting phase. Secondly to use the steady-state data from the Re20 scenario as the initial value. This second possibility is not yet completely supported in the code, so I only used the first one for this time.
Results  After 4000 Pseudo Time Steps we cannot see any ringing any longer. Furthermore, remark the following feature: Here, I used a bigger time-step size for the Pseudo Time Steps than at the first FE run in this chapter. This led to a wrong absolute level of the drag coefficient oscillation - just compare Figure 9. The fascinating point is that TR now revises that level to the right one.

Now we know that there are possible ways to produce good results with TR on time-dependent scenarios, but TR is still very unhandy to use. Hence, the next point is to determine if TR could compete with FE on such problems. On a simple 220x41 grid FE would always be the better choice. But to assure stability on finer grids FE would need a very small time-step size. For example for the 220x41 grid the necessary time-step size is \( \tau = 0.001 \), but the next finer Benchmark Scenario 440x82 needs a five times smaller one: \( \tau = 0.0002 \). Can we observe a similar effect for TR? No. I simulated the 440x41 Benchmark Scenario with TR and a constant time-step size \( \tau = 0.02 \) - just the the same size as for the 220x41 scenario. Look at the good results on Figure 16.
In the final series of test runs, I used TR with an adaptive time-step size. The hope was that this method could handle bad initial values, e.g. Benchmark Scenario without any Pseudo Time Steps. Unfortunately, this is not possible. The initial value is too bad. The adaptive TR methods works similar to the one with a constant time-step size, if you calculate enough Pseudo Time Steps. Then \( \tau \) stays more or less on the same level, because the scenario does not change so much anymore. So I cannot proclaim that an adaptive time-step size is useful for the Benchmark Scenarios, but of course there are scenarios that need an adaptive \( \tau \) - scenarios with sudden changes, but from a physically reasonable initial value. Look on Figure 17 and 18. The median time-step size is here approximately \( 10^{-3} \). For such a small one, you can see the same ringing, again, but not in a bad range.

The bigger 440x82 Benchmark Scenario brought similar plots, but the timestep-size here is approximately \( 10^{-2} \). That means the timestep criterion fits better for a greater number of vertices.

Figure 16: Benchmark Scenario (440x82 Re100), TR: Drag Coefficient, \( \tau = 0.02 \), after 20000 Pseudo Time Steps
Figure 17: Benchmark Scenario (220x41 Re100), TR with adaptive $\tau$: Drag Coefficient

Figure 18: Benchmark Scenario (220x41 Re100), TR with adaptive $\tau$: Lift Coefficient
4.3 Final Conclusion

In this last section I want to give a „general guideline“ - when to use which time-integrator. This should not support some „Black-Box-Thinking“. Every scenario is different. So you should always invest some time and think about the best integrator or at least try different possibilities.

The first thing you should think about before simulating a new scenario is the time-dependence you expect. Should the simulation converge into a steady-state solution or do you expect some oscillating behavior or even a more complicated time-dependency? These are important questions. For the Benchmark Scenarios the difference between steady-state and oscillating is only a higher Reynold’s-Number, but the way you treat these two scenarios is very different as we have seen before.

<table>
<thead>
<tr>
<th>Time-Dependence</th>
<th>Time-Evolution</th>
<th>Meshsize</th>
<th>Integrator</th>
</tr>
</thead>
<tbody>
<tr>
<td>steady-state</td>
<td>unimportant</td>
<td>-</td>
<td>SSS</td>
</tr>
<tr>
<td>steady-state</td>
<td>important</td>
<td>big</td>
<td>FE</td>
</tr>
<tr>
<td>steady-state</td>
<td>important</td>
<td>small</td>
<td>BE</td>
</tr>
<tr>
<td>oscillating</td>
<td>-</td>
<td>big</td>
<td>FE</td>
</tr>
<tr>
<td>oscillating</td>
<td>-</td>
<td>small</td>
<td>TR const</td>
</tr>
<tr>
<td>more complicated</td>
<td>-</td>
<td>big</td>
<td>FE</td>
</tr>
<tr>
<td>more complicated</td>
<td>-</td>
<td>small</td>
<td>TR adapt</td>
</tr>
</tbody>
</table>

Table 12: General Guideline for Time-Integrators

Remark

- **SSS** symbolises the Steady-State Solver.
- If you use **BE** for Steady-State Scenarios you should keep in mind that BE produces some numerical damping. That means the Time-Evolution is accelerated.
- To avoid the ringing of **TR**, you have to allocate a good initial value. Different ways to achieve this are possible.
- **TR adapt** is very sensitive to the accuracy $\epsilon$ and the norm you use.
5 Outlook

5.1 The Peano Grid

As mentioned in the first chapter, the Trivial-Grid should only be a test field. The main goal is to provide implicit ODE-methods for the adaptiv Peano Grid. To achieve this, some further changes are necessary. One problem, probably not the only one, is the following: Our extern non-linear equation software PETSc treats the velocity and pressure values as a huge vector. That is strictly against the Peano philosophy. This vector has to be up and downloaded in every PETSc iteration. That is no good performance. The aim should be to install our own non-linear equation solver, which can handle the local philosophy. I think it should be possible to perform the Newton-iteration locally, but first of all this assumption has to be verified.

5.2 Component Design

The actual design of the Fluid component has grown over a few years. So it is not astonishing that some points are very historical. At first, all implicit calculation is performed two levels below the explicit ones (compare Figure 2). There is no real reason for that. But there is also no simple solution. We should take care of not putting too many functions into the FluidSimulation, as this main class should be clearly arranged. A possible solution could be the one discribed in Figure 19.

As a second point I want to mention the steady-state component. There is no use anymore for that runner classes as a simple steady-state solution could be computed by calculating one BE step with an time step size $\tau = \infty$. So the steady-state case could be treated either as a further son of the ODESolver or simply by changing the configuration.

Figure 19: suggestion for a new fluid design
References


