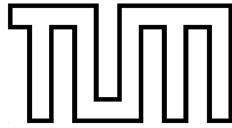


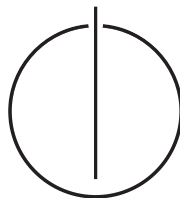
Technische Universität München

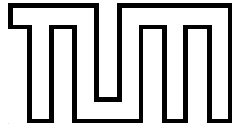
Fakultät für Informatik

Interdisziplinäres Projekt (IDP)

Erweiterung eines stationären Navier-Stokes-Lösers

Asli Okur
Johann Schlamp





TECHNISCHE UNIVERSITÄT MÜNCHEN

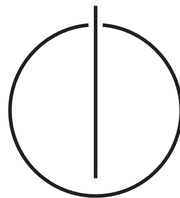
FAKULTÄT FÜR INFORMATIK

Interdisziplinäres Projekt (IDP)

Erweiterung eines stationären Navier-Stokes-Lösers

Extension of a Steady-State Navier-Stokes-Solver

Bearbeiter:	Asli Okur Johann Schlamp
Aufgabensteller:	Prof. Dr. Hans-Joachim Bungartz
Betreuer:	Tobias Neckel
Abgabedatum:	15. November 2008



Contents

1. Introduction **1**
- 1.1. Assignment . . . 1
  - 1.1.1. Initial status of Peano . . . 1
  - 1.1.2. Tasks overview . . . 1
  - 1.1.3. Expectations . . . 2

2. Theoretical Background **3**
- 2.1. Automatic Differentiation . . . 3
  - 2.1.1. Motivation . . . 3
  - 2.1.2. Abstract View . . . 3
  - 2.1.3. Implementation Procedures . . . 5
  - 2.1.4. Tools . . . 6
- 2.2. Analytical Differentiation . . . 7
  - 2.2.1. General Calculation . . . 7
  - 2.2.2. Explicit Example for Unit Test . . . 10

3. Implementation **13**
- 3.1. Modifications for ADOL-C . . . 13
  - 3.1.1. Introduction . . . 13
  - 3.1.2. Example . . . 13
  - 3.1.3. Integration into Peano . . . 13
- 3.2. Implementation of analytical derivative . . . 15
- 3.3. Matrix-Free Considerations . . . 15
  - 3.3.1. Overview . . . 15
  - 3.3.2. Changes to Peano Code . . . 16
  - 3.3.3. Conclusions and Outlook . . . 16
- 3.4. Other Modifications . . . 17
  - 3.4.1. CalculateF . . . 17
  - 3.4.2. CalculateJacobian . . . 17

4. Tests and Experiments **19**
- 4.1. Implementation Tests . . . 19
  - 4.1.1. Unit Tests . . . 19
  - 4.1.2. Integration Tests for Fluid and ADOL-C Components . . . 19
- 4.2. Numerical Experiments . . . 22
  - 4.2.1. DFG-Benchmark . . . 22
  - 4.2.2. Optimizations for ADOL-C . . . 26
  - 4.2.3. Additional Performance Analysis . . . 27

5. Conclusion & Outlook **29**

A. ADOL-C Installation Guide **31**
A.1. Installation 31
A.2. Binding in Eclipse 32

List of Figures **33**

Bibliography **35**

1. Introduction

Numerical simulation generally poses high demands on man and machine. Within the scope of `Peano` - *just another fluid solver* - these demands regularly manifest themselves as limiting factor. Depending on the actual circumstances, "limiting" can have a variety of different meanings. For a start, **technical limitations** always have to be considered. Memory consumption for instance on a level to compete with supercomputers is in most cases out of the question. Furthermore, **time** is ubiquitously considered limited. It definitely makes a difference to run a simulation within three hours instead of three days. Another important constraint will often be **accuracy**. Especially in the field of *nonlinear partial differential equations*, where a butterfly's flapping wing could alter the path of a tornado, **precision** may outweigh the need for speed.

This project focuses as a "*Interdisziplinäres Projekt*" (IDP) on the extension of a steady-state Navier-Stokes solver, in particular with objectives to decrease runtime and memory usage as well as to pull up accuracy.

1.1. Assignment

1.1.1. Initial status of Peano

A preceding IDP already implemented this steady-state solver by using the external PETSc library [1]. The main computational task for the linear solver to calculate Jacobian matrices was rewritten by a *finite differences* approach to better respond to local requirements. Currently, Peano operates on a regular grid, but advancements with adaptive implementations are forging ahead.

1.1.2. Tasks overview

Several tasks in the scope of calculating Jacobian matrices were assigned:

- Evaluation and implementation of an automatic differentiation tool
- Formal calculation and implementation of analytical derivation
- Tuning the existent calculation for the convection and diffusion part of the Navier-Stokes-Equation
- Preparation for matrix-free usage (non-linear solving still with PETSc, but linear solver may be iterative/user implemented)
- Extensive performance tests

1.1.3. Expectations

The IDP's main goals described above will also provide more flexibility. After implementation, there will be three different ways to calculate Jacobian matrices. Each of them should have its advantages, justified by extensive performance analyses. Also, for future extensions like the adaptive grid or multigrid as well as scenarios in 3D, a solid grounding should be established.

Analytical derivation will provide a fast method for calculating actual Jacobian matrices. For more complex functions, where manual derivation would not be possible anymore, automatic differentiation will fit in. Even more future related, if the PETSc linear solver gets to its limits, the basic matrix-free implementation should offer a way to get completely independent of this external library.

Technically, there will also be put focus on reducing and preventing code duplication as well as the intensive use of design patterns like proxies, adapters, facades and templates.

2. Theoretical Background

2.1. Automatic Differentiation

2.1.1. Motivation

The current approach for calculating derivatives respectively Jacobian matrices in the *Peano* project is a specialised method of numerical differentiation by explicitly computing difference quotients. Although this is already a tuned approach, according to detailed performance evaluations (see section 4.2 and the corresponding *doxys pages*) these calculations still take a significant part of the overall cpu-time for computing steady-state solutions. An approach for a more efficient method will be presented in the following.

2.1.2. Abstract View

Automatic differentiation is based on formal differentiation. As every computer program can be decomposed into a sequence of elementary assignments, complex functions can be differentiated by applying the *chain rule* to their elemental partial derivatives. With this approach the cpu-intensive calculations and numerical problems of computing difference quotients are replaced by implementing symbolic transformation at the most basic level. In addition there are two different ways of applying the *chain rule*.

In the following illustrations the function $f(x_1, x_2) = x_1 * x_2 + sin(x_1)$ will be differentiated in both ways.

Forward accumulation

Figure 2.1 shows the basic functionality of forward accumulation. By setting the *seeds* $\dot{w}_1$ and $\dot{w}_2$ to 0 or 1, the derivatives with respect to x_1 or x_2 are distinguished. Starting with $x_1 = w_1$ and $x_2 = w_2$, the final derivative is computed through step-by-step accumulation of already caluclated parts according to the chain rule. Although the table displays a symbolic derivation, it is always the numeric (evaluated) value that is stored.

This implementation is strong for functions $f : \mathbb{R} \rightarrow \mathbb{R}^m, m \gg 1$.

Backward accumulation

Here the seeds determine the partial function, not the part of the argument vector as above. At the bottom nodes the partial derivatives for x_1 and x_2 come out. See Figure 2.2 for details.

This implementation is strong for functions $f : \mathbb{R}^n \rightarrow \mathbb{R}, n \gg 1$.

ForwardAccumulation A concrete implementation of **automatic differentiation** normally doesn't make strictly use of just one of the methods described above. Minimizing the number

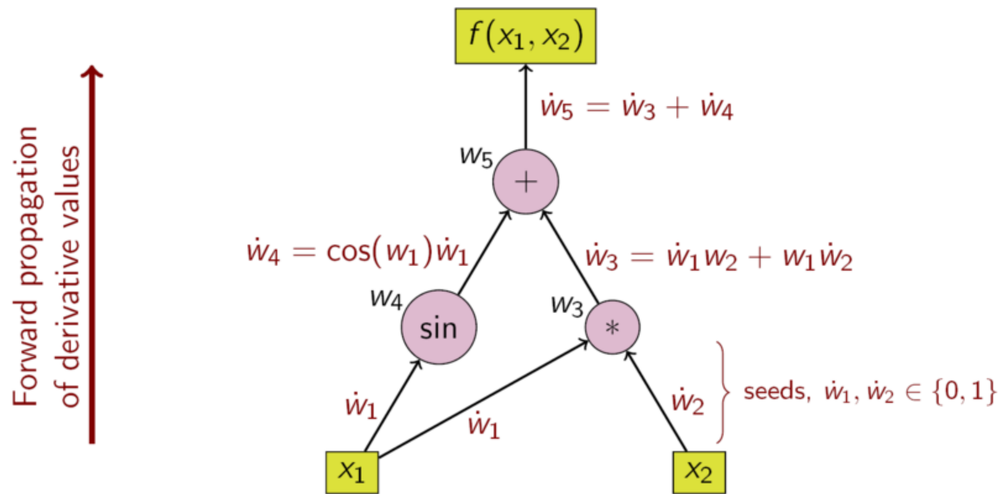


Figure 2.1.: Forward accumulation [2]

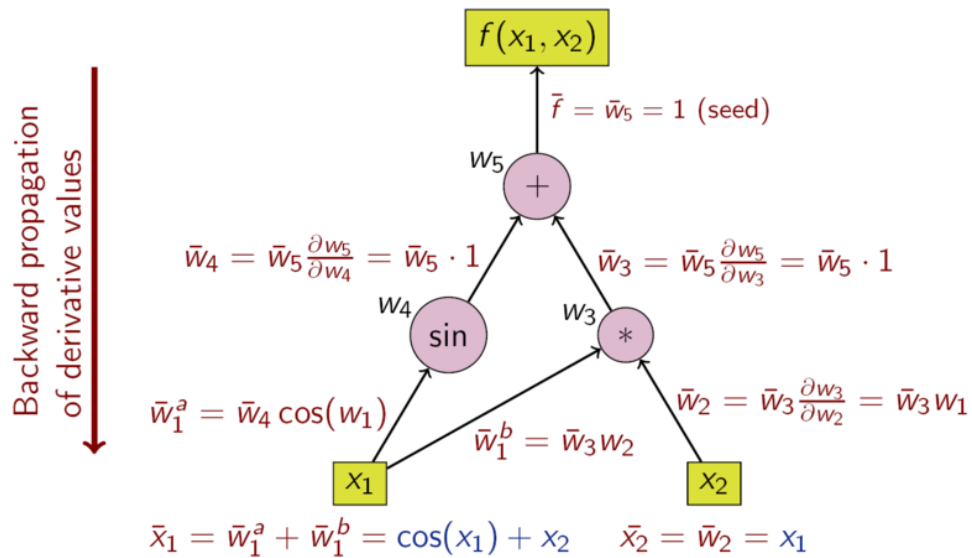


Figure 2.2.: Backward accumulation [2]

of arithmetic operations is known as the *"optimal Jacobian accumulation problem"*. This problem is **NP-complete**.

2.1.3. Implementation Procedures

There is a variety of available tools which do not just differ from theoretical aspects but also from implementation considerations. In fact there are two different ways of providing automatic differentiation functionality.

Source code transformation

As shown in Figure 2.3 the source code for a function is replaced by an automatically generated source code that includes statements for calculating the derivatives interleaved with the original instructions. Source code transformation can be implemented for all programming languages and it is also easier for the compiler to do compile time optimizations. However the implementation of the AD tool itself is more difficult.

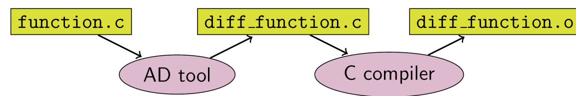


Figure 2.3.: Source code transformation [2]

Operator overloading

The programming language has to support operator overloading. Then objects directly support differentiation and no change in the original source code is required. Operator overloading for forward accumulation is easy to implement, and also possible for reverse accumulation. However, current compilers lag behind in optimizing the code when compared to forward accumulation.
Figure 2.4 shows this simple relationship.

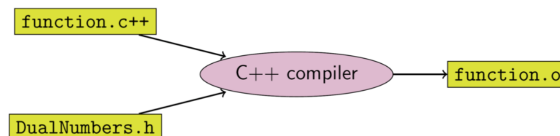


Figure 2.4.: Operator overloading [2]

2.1.4. Tools

The choice for an automatic differentiation package for the Peano project is based on the following table 2.5.

PACKAGE	MODE	VERSION	POSITIVE	NEGATIVE	MISC
ADC	Operator overloading (forward)	unknown	probably efficient	not used widely	commercial
ADIC	Source transformation (forward)	Jun 2005	-	Win32 not supported community dead bad documenation	registration necessary
ADOL-C (to be tested)	**Operator overloading (forward, backward)**	**Aug 2006**	**all OS supported good documentation many features (jacobian, sparsity)**	**-**	**by TU Dresden**
COSY INFINITY	Operator overloading (forward)	2006	all OS supported good documenation suitable for high-dim sparsity features	no community free for private use	registration necessary
CppAD	Operator overloading (forward, backward)	Mar 2008	all OS supproted active community good documentation integrated speed tests sparsity features	-	-
FAD	Operator overloading (forward)	2002	-	Win32 not supported community dead bad documentation	-
FADBAD++	Operator overloading (forward, backward)	unknown	all OS supported documentation ok	no community little information	-
FFADLib	Operator overloading (forward)	unknown	-	Unix not supported	registration necessary
OpenAD	Source Transformation (forward, backward)	Nov 2006	good documentation	Win32 not supported uses external libs	used by NASA
YAO	unknown (forward, backward)	unknwon	-	Win32 not supported no information	website in French

Figure 2.5.: Tools for automatic differentiation

We decided to use **ADOL-C** ("A Package for Automatic Differentiation of Algorithms Written in C/C++"). CppAD looks promising, too, but referring to *Autodiff.org* [3] only one paper's work is based on it, describing most recent advances in automatic differentiation. On the other hand, ADOL-C is widely-spread on different fields like *radiation therapy*, *aero-dynamics* or *circuit simulation.* Additionally, ADOL-C is referred to by almost 50 different papers.

See chapter 3 for a small programming guide on ADOL-C and Appendix A for hints on the installation. For deeper insights and further instructions please refer to the *ADOL-C User's Manual* [4].

2.2. Analytical Differentiation

2.2.1. General Calculation

The starting point is the continuous **Navier-Stokes equations (NSE)** for incompressible flow without the time derivative (as we are interested in the steady state):

$$\begin{aligned} (\mathbf{u}\cdot\nabla)\,\mathbf{u} - \tfrac{1}{\nu}\Delta\mathbf{u} + \nabla p &= \mathbf{0} \\ \nabla\cdot\mathbf{u} &= \mathbf{0} \end{aligned}$$

As for the time-dependent case, we discretise the space with our finite element approach for the velocities (the pressure being a Lagrange multiplier). This results in the following set of **discrete NSE**:

$$B\begin{pmatrix} \mathbf{u}_h \\ p_h \end{pmatrix} := \begin{pmatrix} \underbrace{C(\mathbf{u}_h)\,\mathbf{u}_h + D\mathbf{u_h}}_{=:F} + M^T p_h \\ M\mathbf{u}_h \end{pmatrix} = \mathbf{0}$$

The vector $B(x)$ is also denoted as the **Navier-Stokes function**.

Our main goal here is to implement a method for calculating analytical local Jacobian entries of the **Navier-Stokes function** on one single cell and update the global Jacobian matrix (*PETSc-Matrix*) accordingly. In order to do that we need to study the Jacobian of B on one cell:

$$J_{local} = \left(\begin{array}{c|c} \text{Block 3} & \text{Block 2} \\ \frac{\partial F}{\partial u} & \frac{\partial grad}{\partial p} \\ \hline \text{Block 1} & \\ \frac{\partial div}{\partial u} & 0 \end{array}\right)$$

Important: From now on we work on 2D functions, please note that the calculation is similar for 3D.

Block 1 (`CalculatePPERHS`)

The corresponding function for this block is the following:

$$div(u_a) := Mu_a \text{ with div: } \mathbb{R}^8 \to \mathbb{R}$$

The Jacobian calculation of this block is easy:

$$\frac{\partial div}{\partial u} = M \in \mathbb{R}^{1\times 8}$$

Block 2 (`CalculatePressureGradient`)

The relevant function for this block is:

$$grad(p) := M^T p \text{ with grad: } \mathbb{R} \rightarrow \mathbb{R}^8$$

The calculation of the Jacobian is similar to Block 1:

$$\frac{\partial grad}{\partial p} = M^T \in \mathbb{R}^{8\times 1}$$

Block 3 (`CalculateF`)

This is the main part of the local Jacobian calculation. First, we need to understand how function F works in order to differentiate it (see `CalculateF`).

$$F(u_a) := \underbrace{C(u_a) \cdot u_a}_{bilinear} + \underbrace{D \cdot u_a}_{linear} \text{ with F: } \mathbb{R}^8 \rightarrow \mathbb{R}^8$$

Please note, that this block contains the term $(u * \nabla)u$ and is therefore inevitably more difficult to differentiate.

Linear Part: Matrix D

$$D := \left(\begin{array}{c|c} D^* & 0 \\ \hline 0 & D^* \end{array}\right) \in \mathbb{R}^{8\times 8}$$

Matrix D is the Jacobian of the function $D \cdot u_a$ because it is linear.

Bilinear Part: Matrix C

$$C := \left(\begin{array}{c|c} C^* & 0 \\ \hline 0 & C^* \end{array}\right) \in \mathbb{R}^{64\times 8}$$

The Matrix C^* is a (32×4)-matrix, which consists of 8 (4×4)-matrices. (C_i)

For upper-block:

$$
\begin{array}{cccc}
(u_0^x, u_1^x, u_1^x, u_1^x) & \cdot & C_0 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
(u_0^y, u_1^y, u_1^y, u_1^y) & \cdot & C_1 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
\hline
(u_0^x, u_1^x, u_1^x, u_1^x) & \cdot & C_2 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
(u_0^y, u_1^y, u_1^y, u_1^y) & \cdot & C_3 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
\hline
(u_0^x, u_1^x, u_1^x, u_1^x) & \cdot & C_4 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
(u_0^y, u_1^y, u_1^y, u_1^y) & \cdot & C_5 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
\hline
(u_0^x, u_1^x, u_1^x, u_1^x) & \cdot & C_6 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
(u_0^y, u_1^y, u_1^y, u_1^y) & \cdot & C_7 & \cdot & \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \\
 & & \underbrace{\qquad}_{C^* \in \mathbb{R}^{32\times 4}} & & \underbrace{\qquad}_{\text{for lower-block with } u_i^y}
\end{array}
$$

Part 1: C_i for $i = 0, 2, 4, 6$

$$
A_i := (u_0^x, u_1^x, u_1^x, u_1^x) \cdot \begin{pmatrix} (C_i)_{00} & (C_i)_{01} & (C_i)_{02} & (C_i)_{03} \\ (C_i)_{10} & (C_i)_{11} & (C_i)_{12} & (C_i)_{13} \\ (C_i)_{20} & (C_i)_{21} & (C_i)_{22} & (C_i)_{23} \\ (C_i)_{30} & (C_i)_{31} & (C_i)_{32} & (C_i)_{33} \end{pmatrix}
$$

Part 2: C_i **for** $i = 1, 3, 5, 7$

$$A_i := (u_0^y, u_1^y, u_1^y, u_1^y) \cdot \begin{pmatrix} (C_i)_{00} & (C_i)_{01} & (C_i)_{02} & (C_i)_{03} \\ (C_i)_{10} & (C_i)_{11} & (C_i)_{12} & (C_i)_{13} \\ (C_i)_{20} & (C_i)_{21} & (C_i)_{22} & (C_i)_{23} \\ (C_i)_{30} & (C_i)_{31} & (C_i)_{32} & (C_i)_{33} \end{pmatrix}$$

Part 3: Subfunctions needed for the Jacobian We need to calculate the partial derivatives of functions defined as the following.

For upper-block: **For lower-block:**

$$A_i^x := A_i \cdot \begin{pmatrix} u_0^x \\ u_1^x \\ u_2^x \\ u_3^x \end{pmatrix} \qquad A_i^y := A_i \cdot \begin{pmatrix} u_0^y \\ u_1^y \\ u_2^y \\ u_3^y \end{pmatrix}$$

Result:

$$\frac{\partial F}{\partial u} = \left(\begin{array}{c|cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & \frac{\partial A_0^x}{\partial u_0^x} + \frac{\partial A_1^x}{\partial u_0^x} & \frac{\partial A_0^x}{\partial u_1^x} + \frac{\partial A_1^x}{\partial u_1^x} & \cdots & \frac{\partial A_0^x}{\partial u_3^x} + \frac{\partial A_1^x}{\partial u_3^x} & \frac{\partial A_0^x}{\partial u_0^y} + \frac{\partial A_1^x}{\partial u_0^y} & \cdots & \cdots & \frac{\partial A_0^x}{\partial u_3^y} + \frac{\partial A_1^x}{\partial u_3^y} \\ 2 & \frac{\partial A_2^x}{\partial u_0^x} + \frac{\partial A_3^x}{\partial u_0^x} & \frac{\partial A_2^x}{\partial u_1^x} + \frac{\partial A_3^x}{\partial u_1^x} & \cdots & \frac{\partial A_2^x}{\partial u_3^x} + \frac{\partial A_3^x}{\partial u_3^x} & \frac{\partial A_2^x}{\partial u_0^y} + \frac{\partial A_3^x}{\partial u_0^y} & \cdots & \cdots & \frac{\partial A_2^x}{\partial u_3^y} + \frac{\partial A_3^x}{\partial u_3^y} \\ 3 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 4 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 5 & \frac{\partial A_0^y}{\partial u_0^x} + \frac{\partial A_1^y}{\partial u_0^x} & \cdots & \cdots & \frac{\partial A_0^y}{\partial u_3^x} + \frac{\partial A_1^y}{\partial u_3^x} & \frac{\partial A_0^y}{\partial u_0^y} + \frac{\partial A_1^y}{\partial u_0^y} & \cdots & \cdots & \frac{\partial A_0^y}{\partial u_3^y} + \frac{\partial A_1^y}{\partial u_3^y} \\ 6 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 7 & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 8 & \frac{\partial A_6^y}{\partial u_0^x} + \frac{\partial A_7^y}{\partial u_0^x} & \cdots & \cdots & \frac{\partial A_6^y}{\partial u_3^x} + \frac{\partial A_7^y}{\partial u_3^x} & \frac{\partial A_6^y}{\partial u_0^y} + \frac{\partial A_7^y}{\partial u_0^y} & \cdots & \cdots & \frac{\partial A_6^y}{\partial u_3^y} + \frac{\partial A_7^y}{\partial u_3^y} \end{array} \right) + D$$

Notes:

- Don't forget to add the matrix D. (See "Linear Part" section 2.2.1)
- For simpler documentation, `hFactor` (vector h determines the size of the cell on the grid) is not explained here, please consider this during calculation or implementation. (For more details see doxys-pages [5] or source code explanations)

2.2.2. Explicit Example for Unit Test

In order to test our approach of Jacobian calculation, we created a simple test case and calculated every single entry of the local Jacobian manually. Later on we implemented this scenario as an integration test for the steady-state component and also as a unit test for `CalculateF` in `fluid`. (See also section 4.1)

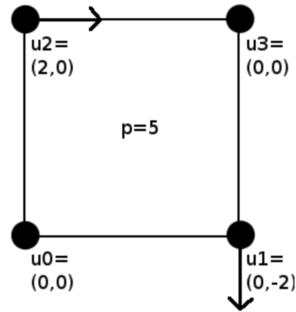


Figure 2.6.: Basic test scenario

According to this scenario we have the following input values:

$$B(x) = B \begin{pmatrix} 0 \\ 0 \\ 2 \\ 0 \\ 0 \\ -2 \\ 0 \\ 0 \\ 5 \end{pmatrix} \text{ with } h = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

The manually calculated local Jacobian would have the dimensions 9×9. After applying the calculations described in subsection 2.2.1 we get the following Jacobian matrix. These results have been used later on for testing correct implementation of all three differentiation methods.

$$\begin{pmatrix} 5/9 & 0 & -1/2 & -7/18 & 1/3 & 1/9 & 1/6 & 1/18 & 1 \\ -1/6 & 17/18 & -5/9 & -5/9 & 1/9 & 1/9 & 1/18 & 1/18 & -1 \\ -1/3 & -2/9 & -1/18 & -1/18 & 1/6 & 1/18 & 1/3 & 1/9 & 1 \\ -7/18 & -1/18 & -5/9 & 1/3 & 1/18 & 1/18 & 1/9 & 1/9 & -1 \\ -1/3 & -1/6 & -1/9 & -1/18 & 7/9 & 1/6 & -1/3 & -5/18 & 1 \\ -1/6 & -1/3 & -1/18 & -1/9 & 0 & 25/18 & -4/9 & -5/18 & 1 \\ -1/9 & -1/18 & -1/9 & -1/18 & -1/6 & -1/9 & 7/18 & 2/9 & -1 \\ -1/18 & -1/9 & -1/18 & -1/9 & -5/18 & 2/9 & -5/18 & 1 & -1 \\ -1 & 1 & -1 & 1 & -1 & -1 & 1 & 1 & 0 \end{pmatrix}$$

3. Implementation

3.1. Modifications for ADOL-C

3.1.1. Introduction

Three principal steps are necessary for using ADOL-C's automatic differentiation implementation:

1. The variables of ADOL-C type `adouble` have to be defined.
2. The method to differentiate has to be declared within the `trace()` statement so that the function will be recorded on a "`tape`".
   - The input values need to be activated with operator <<=
   - Actual calculation of the function needs to be done with adouble type within the `trace()`-block.
   - The output values need to be specified with operator >>=
3. ADOL-C methods like `gradient()` or `jacobian()` can be used.
   - Please note that these methods need `double*` pointers as input and output values.

3.1.2. Example

Figure 3.1 contains example code for the simple function $f(x) = x^2$. More complicated examples can be found in the *ADOL-C User's Manual* [4].

Notes:

- You can specify more than one function with ADOL-C. All you need to do is to start the `trace()`-block with `trace_on(i)` where `i` is the number of your tape.
- If you have higher-dimensional functions, you need to be careful with the order of activation and deactivation. It is not the order stored in the input and output arrays, but the order you recorded them on tape.

3.1.3. Integration into Peano

The local Jacobian matrices get computed in the `CalculateJacobian` class. First, the calculation of the actual function B consisting of contributions from F, the divergence and the pressure gradient is defined within the `trace()`-block. After that, a call to `jacobian()` with predefined working data obtained from the grid and an initialized output array provides the local Jacobian matrix. Then these values get added to the global Jacobian matrix with the help of specific precalculated indices.

```
double *x = new double[1];
adouble *xa = new adouble[1];

double y;
adouble ya;

double *result = new double[1];

x[0] = 3.0;

trace_on(1);  // Creating tape 1
      xa[0] <<= x[0];
      ya = xa[0]*xa[0];
      ya >>= y;
trace_off();

gradient(1,1,x,result);  // result[0] is now 6.0

x[0] = 2.0;
gradient(1,1,x,result);  // result[0] is now 4.0
```

Figure 3.1.: Simple example for ADOL-C code

The most important step is the correct function definition, the rest is done automatically. As all functions used with ADOL-C have to use the data type *adouble*, the corresponding `compute`-methods in `CalculateF`, `CalculatePressureGradient` and `CalculateDivergence` had to be modified, too. This was realized with `ifdefs` and lead to some unavoidable code duplication, but if not using ADOL-C the standard data type *double* is the better choice. See the numerical experiments in section 4.2 for an overview of performance losses when compiled with ADOL-C but using other differentiation methods.

For further technical details please refer to the corresponding header files or the *doxys-pages* [5].

3.2. Implementation of analytical derivative

As first step of this procedure we implemented the analytical derivative function for 2D.

According to the information provided in section 2.2 we had to modify the three classes `CalculateF`, `CalculatePressureGradient` and `CalculatePPERHS` so that they all have new methods for calculating the local analytical Jacobian entries. We also added a new method for analytical differentiation in the `CalculateJacobian` class and calculated the local Jacobian `_elementJacobian` of one cell these three functions mentioned above.

Please note that these three new methods are independent of vertex-type, so the consideration of vertex-type is done just before the `addJacobianBlock` call in the new analytical method of `CalculateJacobian` class, where we just initialize the indices of non-degree-of-freedom-vertices with `_ignoreIndexValue` in the adress-array `_elementJacobianRowsCols` (used in `addJacobianBlock`).

Later on, we compared the results with other two methods for differentiation by creating some unit and integration test cases (see section 4.1 for test details).

After getting matching results in test cases, we wanted to implement this function also for 3D. In order to do that, we first had to unify the `CalculateF`-calculation in general, so that we could use one generic calculation method in both 2D and 3D simulations. This also prevented code duplication (for further details of this step, see section 3.4). Lastly, we adopted our 2D analytical differentiation approach in dimensionless form.

During our tests and performance runs (see section 4.2) we determined that this part of our work was very efficient, because we got very good Jacobian-assembly times compared to the other two methods.

For further details about these modifications please refer to *doxys-pages* [5] or source code explanations/comments of relevant classes.

3.3. Matrix-Free Considerations

3.3.1. Overview

For complex 2D scenarios and most of the 3D scenarios, the preconditioner ILU consumes too much main memory. Even test runs on a machine with 16 Gb of RAM fail due to lack of memory. In this context, the idea of a matrixfree calculation is born.

In principal, the Krylov linear solver has to compute a matrix-vector product $y = J * x$ in every linear iteration. J refers to the precedingly calculated global Jacobian matrix. The new matrixfree procedure waives this calculation. Instead, the computation of the global output vector y happens directly and completely within the multiplication method. This means, no full scale matrix-vector product has to be computed (and therefore no oversized matrix has to be held in memory), but rather many elementary products with local input vectors x^e on each cell, which then get added to the global output vector.

There are several things to be taken care of. First, the PETSc nonlinear solver has to be “faked” by an empty method making it “believe” that the global Jacobian matrix has already been computed. Subsequently, the PETSc internal matrix multiplication method **MatMult** has to be overwritten in order to not only calculate a global product but in fact to iterate over the grid, compute every local Jacobian contribution and multiplying it with the corresponding part of the input vector. This *corresponding part* has to be of some concern, too.

3.3.2. Changes to Peano Code

In order to understand the new behaviour of the PETSc part in Peano, a fundamental knowledge of the nonlinear solver *SNES* and the linear solver *KSP* is recommended. See *PETSc User's Manual* [6] for further details.

For a start most parts of the steady state calculation framework had to be modified. Starting in `AbstractSteadyStateSolver`, a boolean flag determining the use of matrixfree calculations gets read from the config-file, starting a slightly different setup. For the matrixfree case, the`(void*)` pointers for the new matrix multiplication method and the fake Jacobian calculation method have to be set in `PETScNonLinearSolver`. These methods are located in `PETScNonLinearEquations`. Additionally, the wrapper classes' *solve*-method now gets called with the additional boolean flag.

There, the user-provided Jacobian calculation method gets set through `SNESSetJacobian`, which for the matrixfree case provides the pointer to a nearly empty method `jacobianGridMF`. Except some logging statement, the only task is to obtain and store the pointer for the input vector x after each nonlinear iteration. As all of the Jacobian calculation methods fetch the input x (which in fact are the velocities u and the pressures p) directly from the grid, it is necessary to upload this data to the grid for every new linear iteration. This happens in `AbstractSteadyStateSolver::computeNavierStokesJacobianMF` initially called by the new matrix multiplication method.

Getting back to the wrapper's *solve*-method, there is a change for the matrix creation as well. For the matrixfree case, another type of matrix has to be initialized: `MAT_SHELL`. This one allows overwriting the standard operators as well as defining some own data structure. Providing this so-called *matrix context* through `SNESSetJacobian`, the matrixfree calculation is almost complete.

Last thing to do is to modifiy the linear solver's setup method now also featuring an additional boolean matrixfree flag. For the matrixfree case, the linear solver has to be set up for not using a preconditioner as it is not possible to multiply this full matrix to the matrixfree context. This issue leads to practical problems as described in the next section.

3.3.3. Conclusions and Outlook

The basic implementation for the new matrixfree mode is complete and operating as designed.

Unfortunately the procedure described above doesn't allow the use of standard preconditioners. This results not only in a not-converging linear (and therefore also nonlinear) solver as it would also be the case for the standard calculation without using any preconditioner, but also in a wrong calculation of PETSc's residual norm. Generally it gets computed by a recursive procedure, but after each 30th step the calculation is done exactly: $res = ||y - J * x||$. With no preconditioner in use, there is a large *drift* between the residuals at this step and the one before, so the effect of gaining accuracy is lost at each 30th iteration.

This could be solved by implementing a matrixfree preconditioner as well or with the use of *multigrid*, but this is no more part of our project.

See the *doxys-pages* [5] and especially the corresponding header files for additional information.

3.4. Other Modifications

3.4.1. CalculateF

In the past the calculation of F was optimized for the divergence-free case, but used for the dlinear case, too (in 2D). The 3D method was generic.

For optimizing the standard case, the 3D method has been enhanced for 2D and the div-free method has been shifted to an inherited class `CalculateFDivFree`. As these different calculation methods use different element matrices, a new method `copyMatrices` has been implemented and accordingly overwritten in the child class. During this step a bug in the generic method has been found and fixed.

The performance gains can be evaluated in section 4.2.3.

3.4.2. CalculateJacobian

Another major change to the Peano code was the rewriting of the `CalculateJacobian` class. Meanwhile there are six different ways for calculating the Jacobian matrix. Those can be chosen through the config-file statement *jacobian-assembly-type* with valid types `FINITEDIFF`, `ADOLC` and `ANALYTICAL`. Also, the use of matrix-free calculations can be determined like that.

To prevent writing six full methods, the calculation was splitted into *writingToPETSc* and *computational* functions. According to the config flags the corresponding submethods get called, downsizing the actual code.

The slight performance influence of this new proxy design (a few more additional function calls) is analyzed in section 4.2.3.

4. Tests and Experiments

4.1. Implementation Tests

4.1.1. Unit Tests

ADOL-C

After binding ADOL-C component correctly to Peano and studying the accordant *ADOL-C Manual*[4] pages, we first created two test cases for checking library functionality, where we implemented some examples explained in the manual and a simple Jacobian calculation, so that we could understand the principles of programming with ADOL-C and warm up for more complicated functions like `CalculateB`. See *doxys-pages* [5] and especially the corresponding header file for technical details.

Fluid

As described in subsection 2.2.2 we created a unit test for checking correct implementation of analytical differentiation method.

PETSc

Before implementing the whole matrixfree framework, some proof of the basic approach was necessary. For that, we implemented a set of classes reflecting the behaviour of Peano respectively the calculation within the nonlinear solver. The key requirement was to test the new matrix framework with a customized `MatMult` method calling some external methods simulating an iteration over the grid.

These tests were promising, so the matrixfree procedures got implemented. As these required major changes to high-level classes like `AbstractSteadyStateSolver`, additional integration tests are hardly possible to realize. A whole scenario including instances of all relevant classes would have had to be recreated, which wasn't worth the effort as it is more feasible to just start a regular scenario from a config-file for simple testing purposes. The PETSc integration tests on the other hand are already existing, so the next section only focuses on fluid and ADOL-C components.

4.1.2. Integration Tests for Fluid and ADOL-C Components

Basic generic test

For testing all three differentiation methods basically, we created a *generic* test method based on one single cell. The scenario is the same as the one we used for testing correct calculation of analytical `CalculateF` values (see subsection 2.2.2). All velocities are zero except the vertical one of node 1 (-2.0) and the horizontal one of node 2 (+2.0). The pressure on the cell is set to 5.0.

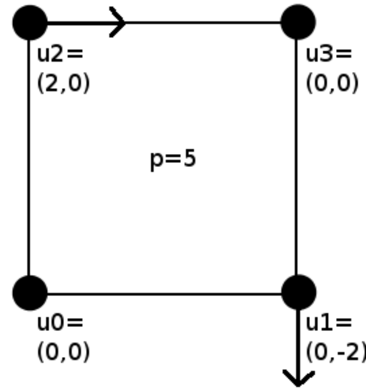


Figure 4.1.: Basic test scenario

This method works for both 2D and 3D and can be run with assembling types `FINITEDIFF`, `ADOLC` and `ANALYTICAL`, where the results will be compared to analytical calculation (without direct use of `CalculateJacobian` class).

Using this method, we also found out that stretched meshing is not yet supported for `FINITEDIFF`, but `ADOLC` and `ANALYTICAL` are implemented accordingly. (See source code comments for further details)

Advanced generic test

Testing only one single cell would not be enough to test correct interaction of fluid, ADOL-C and PETSc components, so we extended our test, where we have 4 cells, 6 inner and 3 Dirichlet vertices for 2D case. The cells are ordered in 2×2, the vertices numbered in lexicographic manner, the left edges are Dirichlet vertices. The pressures in the cells are set to 3.0, 5.0, 2.0, 4.0, respectively and velocities are random as in Figure 4.2, where every single component of velocity vectors is either 0 or ± 1.

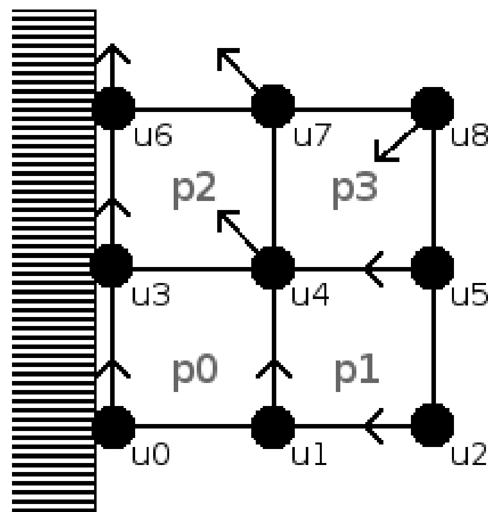


Figure 4.2.: Advanced test scenario

The global Jacobian PETSc matrix has the dimension 22×22 for the 2D case. Our goal by implementing this test is to check if the global Jacobian matrix is updated with correct local Jacobian entries and also if the local Jacobian values for Dirichlet vertices are ignored correctly by updating the global matrix..

This method compares the results of two of three calculation methods. Please note again that for comparing `FINITEDIFF` with one another, all components of the vector `h` should have the same value (corresponding to quadratic cells), because the finite differences method does not support stretched grid yet.

Furthermore, we tried to adopt this test for 3D, where we have a $2 \times 2 \times 2$ scenario with again all left edges are Dirichlet vertices. The nine vertices in front are the same as in 2D (indices 0-8), the others are all set to zero. The cells in the back have all the pressure 1.0. This method might need some improvements, maybe a more logical scenario, but our first concern here was just to see if these three methods are integrated properly in 3D. During this procedure we also found a bug in divergence calculation in `FINITEDIFF`.

4.2. Numerical Experiments

4.2.1. DFG-Benchmark

This benchmark simualates the fluid flow around a cylinder in a canal as described in [7]. Figure 4.3 shows an examplary result for the `440x82` grid.

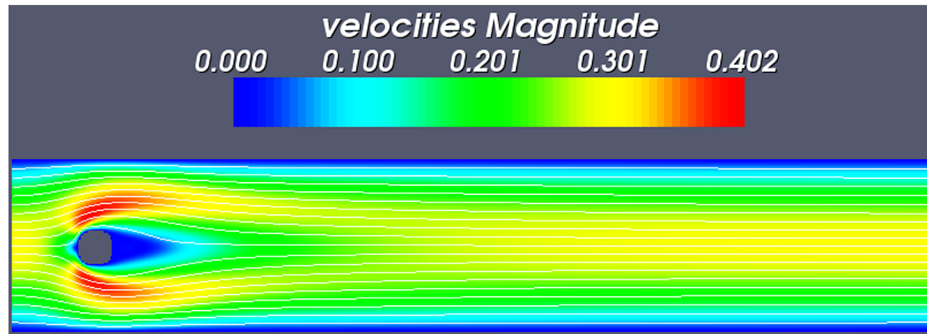


Figure 4.3.: DFG-Benchmark

General information

- All tests are based on the **GMRES** linear solver with a relative convergence tolerance of $1e^{-7}$.
- We used **ILU** as preconditioner for the linear solver.
- The given number of steps represents the number of pseudo-timesteppings.
- Without pseudo-timestepping (*-1* in config-file), the linear solver diverged.
- Performance runs in 3D are not possible due to lack of memory on the practicant's computers.
- The gprof output runs were separately compiled. For the performance and memory analyses we compiled without gprof output (no *-p -pg*).
- The memory usage has been measured with Peano's built-in tool. Alternatives like Linux's **top** provide the same relative results, although the overall usage is slightly smaller.

Test environment: Intel(R) Pentium(R) D CPU 2.80GHz, 3 GB RAM

Compile-flags: Release-mode (*-DDim2 -DDloopOptimiseAggressive -DPackedRecords -DLogTrace -DLogMessageType -DLogTime -DMPICH_IGNORE_CXX_SEEK -O3*)

Tests compiled without ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
FINITEDIFF					
2 steps ILU	41s	0.95s	418	112 - 98 - 113 - 95	104 MB
10 steps ILU	41s	0.96s	378	108 - 92 - 89 - 89	104 MB
ANALYTICAL					
2 steps ILU	37s	0.16s	410	105 - 91 - 112 - 96	114 MB
10 steps ILU	37s	0.15s	362	102 - 93 - 85 - 82	114 MB

Tests compiled with ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
ADOLC					
2 steps ILU	43s	1.28s	410	105 - 91 - 112 - 96	105 MB
10 steps ILU	45s	1.28s	362	102 - 93 - 85 - 82	105 MB
FINITEDIFF					
2 steps ILU	72s	8.05s	418	112 - 98 - 113 - 95	104 MB
10 steps ILU	72s	7.99s	378	108 - 92 - 89 - 89	104 MB
ANALYTICAL					
2 steps ILU	40s	0.16s	410	105 - 91 - 112 - 96	114 MB
10 steps ILU	40s	0.17s	362	102 - 93 - 85 - 82	114 MB

Figure 4.4.: Results for 220x41 test scenario

Results:

Conclusions drawn from the performance runs:

- For the steady-state Navier-Stokes-Solver, automatic differentiation with ADOL-C does not have any overall performance gains. This could be different for other deployments in the future though.
- Depending on the scenario, the **overall performance gain** for the analytical Jacobian calculation is up to **3%** in exchange for **11%** of additional memory usage.
- If compiled with ADOL-C, the **overall performance losses** for
  - the finite differences Jacobian calculation is **heavy** (because of intense use of `adouble` data type)
  - the analytical Jacobian calculation is **negligible**
- The Jacobian matrix assembly time for itself on the other hand has been improved dramatically for analytical derivation (up to **665%**)

Tests compiled without ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
FINITEDIFF					
2 steps ILU	434s	3.90s	1888	474 - 446 - 447 - 521	355 MB
10 steps ILU	509s	4.27s	2353	480 - 529 - 566 - 778	355 MB
100 steps ILU	533s	4.42s	2146	663 - 453 - 421 - 609	355 MB
ANALYTICAL					
2 steps ILU	425s	0.63s	1916	599 - 367 - 382 - 568	394 MB
10 steps ILU	495s	0.64s	2354	601 - 432 - 557 - 764	394 MB
100 steps ILU	523s	0.61s	2177	669 - 455 - 423 - 630	394 MB

Tests compiled with ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
ADOLC					
2 steps ILU	454s	5.00s	1916	599 - 367 - 382 - 568	356 MB
10 steps ILU	527s	4.96s	2342	601 - 432 - 557 - 752	356 MB
100 steps ILU	638s	5.07s	2177	669 - 455 - 423 - 630	356 MB
FINITEDIFF					
2 steps ILU	526s	33.12s	1680	426 - 270 - 431 - 553	355 MB
10 steps ILU	651s	32.55s	2494	699 - 554 - 651 - 590	355 MB
100 steps ILU	761s	33.60s	2133	663 - 453 - 421 - 596	355 MB
ANALYTICAL					
2 steps ILU	431s	0.65s	1916	599 - 367 - 382 - 568	394 MB
10 steps ILU	502s	0.64s	2327	601 - 432 - 557 - 737	394 MB
100 steps ILU	633s	0.63s	2177	669 - 455 - 423 - 630	394 MB

Figure 4.5.: Results for 440x82 test scenario

Inconsistencies:

- The additional memory usage for the analytical calculation cannot be explained. The code has been examined for memory leaks, but there are none. A possible explanation could be four interleaved `for`-loops in `CalculateF`, but that hasn't been proofed.
- It is very confusing that the linear iteration counts in the 440x82 scenario differ for analytical calculation compiled with and without ADOLC. As this code is definitely independent from ADOLC, we don't have any reasonable explanation.
- The different linear iteration counts on the other hand for the finite differences compiled with and without ADOLC could be based on the use of the data type `adouble`, where the calculations possibly aren't performed at double precision. The ADOLC manual [4] only gives small hints on that fact, so a definite statement can't be made.

Tests with more pseudo-timestepping

In contrast to the 220x41 Cylinder scenario, more pseudo-timesteps in 440x82 Cylinder scenario do not automatically reduce the number of linear iterations or overall runtime. This results from smaller timesteps for the finer grid, so better accuracy takes effect just after more steps.

The following tables shows data for better comparison of the pseudo-timestepping's effect on overall time and linear iterations.

Tests compiled without ComponentAdolC

	Time for pseudo-steps	Overall time	Linear iterations
FINITEDIFF			
2 steps ILU	84s	434s	1888
10 steps ILU	89s (+6%)	509s (+17%)	2353 (+25%)
100 steps ILU	141s (+68%)	533s (+22%)	2146 (+14%)
ANALYTICAL			
2 steps ILU	84s	425s	1916
10 steps ILU	88s (+5%)	495s (+16%)	2354 (+23%)
100 steps ILU	143s (+62%)	523s (+23%)	2177 (+14%)

Tests compiled with ComponentAdolC

	Time for pseudo-steps	Overall time	Linear iterations
ADOLC			
2 steps ILU	86s	454s	1916
10 steps ILU	98s (+14%)	527s (+16%)	2342 (+22%)
100 steps ILU	229s (+166%)	638s (+40%)	2177 (+14%)
FINITEDIFF			
2 steps ILU	86	526s	1680
10 steps ILU	98s (+14%)	651s (+23%)	2494 (+48%)
100 steps ILU	231s (+168%)	761s (+45%)	2133 (+27%)
ANALYTICAL			
2 steps ILU	86s	431s	1916
10 steps ILU	98s (+14%)	502s (+16%)	2327 (+21%)
100 steps ILU	234s (+172%)	633s (+47%)	2177 (+14%)

Figure 4.6.: Comparison of pseudo-timestepping for 440x82 test scenario

Conclusions:

- If ADOL-C is compiled, the time needed for pseudo-steps increases dramatically (effecting overall runtime, too), as the pseudo-timesteps now have to be computed with data type `adouble`.
- More steps reduce the number of linear iterations, but only after a certain threshold. There's a peak here at 10 steps.
- The additional time for more pseudo-steps do not influence the overall runtime in the same proportion as more steps result in a better starting value.
- If you want to reduce linear iterations, just increase the number of timesteps to a very high value.
- If you want to reduce overall runtime, use only few timesteps and the analytical mode. In the future, automatic differentiation mode might be used especially for complexer scenarios after some optimizations.

4.2.2. Optimizations for ADOL-C

When we chose ADOL-C as our automatic differentiation tool, we did not know about the speed problem, there was not much information available on the web at the first sight and our first concern was to have an automatic differentiation implementation as an alternative working properly. After getting such results, we searched for answers more deeply and we found some papers and documentations about enforcing ADOL-C's performance ("Run time optimization was not the primary aspect of the ADOLC development!" [8]).

Furthermore according to the *ADOL-C Manual* [4], array initializations like

```
adouble* my_array[size];
```

dramatically decrease performance. So we removed them completely and replaced them with `new` statements in the constructor and `delete` statements in the destructor. The result is shown below.

Note: There are no tests necessary for the analytical differentiation method as it doesn't make use of the data type `adouble`.

Tests compiled with ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
ADOLC					
2 steps ILU	43s	1.28s	410	105 - 91 - 112 - 96	105 MB
10 steps ILU	45s	1.28s	362	102 - 93 - 85 - 82	105 MB
FINITEDIFF					
2 steps ILU	72s	8.05s	418	112 - 98 - 113 - 95	104 MB
10 steps ILU	72s	7.99s	378	108 - 92 - 89 - 89	104 MB

Tests after tuned calculation methods with ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
ADOLC					
2 steps ILU	44s	1.34s	410	105 - 91 - 112 - 96	105 MB
10 steps ILU	45s	1.34s	362	102 - 93 - 85 - 82	105 MB
FINITEDIFF					
2 steps ILU	67s	7.15s	418	112 - 98 - 113 - 95	104 MB
10 steps ILU	69s	7.16s	378	108 - 92 - 89 - 89	104 MB

Figure 4.7.: Results for 220x41 test scenario after ADOL-C optimizations

Conclusions:

- It definitely improves overall runtime and Jacobian assembling time for the finite differences approach, but not as much as expected.
- ADOL-C gets slightly slower (as we have to copy the results now into the return array).
- Another way would be to separate the actual compute-methods completely for FINITEDIFF and ADOLC, but our earlier task was to consolidate them for all different types of Jacobian assembly.

Presumably ADOL-C's Operator Overloading can be held responsible for the bad performance when calculating with `adouble`. So ADOL-C should only be compiled, if automatic differentiation is really needed, otherwise you can expect huge performance losses. Methods like tapeless forward differentiation [8] might improve ADOL-C functionality in Peano in the future.

4.2.3. Additional Performance Analysis

A. Optimizations for CalculateF

After the optimizations in `CalculateF`-class described in section 3.4 we were able to get **better** overall performance (up to **9.9%**), but **2-3%** additional memory usage.

The following table shows the overall time needed for our new method `computeCandD` determined by the gprof output. This method computes the convection and diffusion terms and gets called one time per time step on each cell. For steady-state solutions it is also used for the function evaluation as well as for the Jacobian calculation by finite differences or ADOLC.

Test environment: Intel(R) Pentium(R) D CPU 2.80GHz, 3 GB RAM

Compile-flags: Release-mode (*-DDim2 -DDloopOptimiseAggressive -DPackedRecords -DLogTrace -DLogMessageType -DLogTime -DMPICH_IGNORE_CXX_SEEK -O3*)

	`computeCandD()` time	Memory used
220x41, old version (average)	26.49s	36Mb
220x41, tuned (no adapters) (average)	23.46s	37Mb
220x41, tuned (average)	24.10s	37Mb

Figure 4.8.: Results of `CalculateF` performance runs

Notes:

- Peano is compiled with very few components, just the necessary ones.
- If compiled with AdolC, **performance losses** up to **50%** are unavoidable as a result of intense use of data type `adouble`.
- There is also a test run in the SVN without the use of adapters for `CalculateF`.
  Conclusion: No performance loss for using adapters.
- Because of trouble while migration to the new SVN, the old test runs haven't been compiled with the flag *DLoopOptimiseAggressive* and can't be recompiled. So only absolute times of the tuned method are listed as the overall times are not comparable any more. Just for personal interest, the following table shows the impact of this compile flag on the overall runtime.

	220x41	440x82
No optimization flag	1156s	9399s
With *DLoopOptimiseAggressive*	836s	6939s

Figure 4.9.: Effect of optimization flag

B. Modifications for CalculateJacobian

For the matrix-free implementation, another three methods in `CalculateJacobian` were necessary. To improve structure and readability, we implemented six proxy functions which just call their corresponding calculation method and the right PETSc-write proxy function.

This lead to slightly slower runtimes (~**2.5%** overall and **4%** for Jacobian assembly).

Tests compiled without ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
FINITEDIFF					
(OLD)10 steps ILU	41s	0.96s	378	108 - 92 - 89 - 89	104 MB
(NEW)10 steps ILU	45s	1.02s	378	108 - 92 - 89 - 89	104 MB
ANALYTICAL					
(OLD)10 steps ILU	37s	0.15s	362	102 - 93 - 85 - 82	114 MB
(NEW)10 steps ILU	38s	0.15s	362	102 - 93 - 85 - 82	114 MB

Tests compiled with ComponentAdolC

	Overall time	Average Jacobian assembling time	Linear iterations *overall*	Linear iterations *per nonlinear iterations*	Memory used
ADOLC					
(OLD)10 steps ILU	45s	1.28s	362	102 - 93 - 85 - 82	105 MB
(NEW)10 steps ILU	46s	1.32s	362	102 - 93 - 85 - 82	105 MB
FINITEDIFF					
(OLD)10 steps ILU	72s	7.99s	378	108 - 92 - 89 - 89	104 MB
(NEW)10 steps ILU	73s	8.10s	378	108 - 92 - 89 - 89	104 MB
ANALYTICAL					
(OLD)10 steps ILU	40s	0.17s	362	102 - 93 - 85 - 82	114 MB
(NEW)10 steps ILU	41s	0.16s	362	102 - 93 - 85 - 82	114 MB

Figure 4.10.: Influence of the new proxy functions on 220x41 test scenario

5. Conclusion & Outlook

To conclude this project, a few retrospective annotations will be given.

After investing great effort to familiarize with `Peano`, the benefit of the given assignment was clearly foreseeable and the expectations made in chapter 1 are mostly fulfilled to the full extent.

Technically, the code has been optimized in many respects, reducing code duplication and making consequent use of various design patterns. Apart from this paper, corresponding doxys-pages and detailed header comments were composed.

The integration of an automatic differentiation tool has been completed and validated carefully. Unfortunately the performance gain was not as high as expected, but a *tapeless* approach for future extensions looks promising. Analytical derivation indeed improved runtimes as well as accuracy just as predicted. This differentiation method should from now on be the first choice. For more complex scenarios, where manual derivation may not be possible anymore, automatic differentiation will certainly have its advantages. Additionally, many performance runs with detailed analyses supported these conclusions.

Another promising part are the matrix-free calculations. The necessary framework is fully implemented and ready to use. Still, there are convergence problems as it is no more possible to use a standard preconditioner. This could be the starting point for another student project, maybe also with completely replacing the PETSc linear solver by a user-provided full matrix-free version.

A. ADOL-C Installation Guide

A.1. Installation

Here are the steps you should get through:

- Download the the ADOL-C package for your system from the ADOL-C website.
  (Version: 1.10.2)
- Extract the contents of the package on your hard drive in your home directory.
  Command: `tar -xzf adolc-1.10.2.tar.gz ~`
- Execute configure-script (~1 min):
  Command:
  ```
  cd ~/adolc-1.10.2
  ./configure
  ```
- Execute make-command (~2 min):
  Command: `make`
- Execute make all (~10 sec):
  Command: `make all`
- Add these to lines in your `~/.bash_profile`
  (If `~/.bash_profile` does not exist, create it):
  ```
  LD_LIBRARY_PATH=/home_local/USERNAME/adolc_base/lib:/home_local/USERNAME/adolc_base
  export LD_LIBRARY_PATH
  ```

Notes:

- The last output of configure script is useful for section A.2
- If you don't want to log out soon, before using the library for the first time, you might have to type in "`source ~/.bash_profile`".
- **For KDE users:** `~/.bash_profile` might not be executed in login process. In that case you might have to type in: "`source ~/.bash_profile`" everytime if you get the error: *"error while loading shared libraries: libadolc.so.0: cannot open shared object file: No such file or directory"*

A.2. Binding in Eclipse

You need to add these settings in your project:

C++ Compiler options	
Include (-I)	`/home_local/USERNAME/adolc_base/include`
C++ Linker options	
Libraries (-l) Library search path	`adolc` `/home_local/USERNAME/adolc_base/lib`

Notes:

- As described in former section A.1, you need to add these to lines in your `~/.bash_profile` (If ~/.bash_profile does not exist, create it):
  ```
  LD_LIBRARY_PATH=/home_local/USERNAME/adolc_base/lib:/home_local/USERNAME/adolc_base
  export LD_LIBRARY_PATH
  ```
- For further help please refer to ADOL-C Homepage.

List of Figures

2.1. Forward accumulation [2] 4
2.2. Backward accumulation [2] 4
2.3. Source code transformation [2] 5
2.4. Operator overloading [2] 5
2.5. Tools for automatic differentiation 6
2.6. Basic test scenario 11

3.1. Simple example for ADOL-C code 14

4.1. Basic test scenario 20
4.2. Advanced test scenario 20
4.3. DFG-Benchmark 22
4.4. Results for 220x41 test scenario 23
4.5. Results for 440x82 test scenario 24
4.6. Comparison of pseudo-timestepping for 440x82 test scenario 25
4.7. Results for 220x41 test scenario after ADOL-C optimizations 26
4.8. Results of `CalculateF` performance runs 27
4.9. Effect of optimization flag 28
4.10. Influence of the new proxy functions on 220x41 test scenario 28

Bibliography

[1] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc Web page," 2001. http://www.mcs.anl.gov/petsc.

[2] J. Wales and L. Sanger, "Wikipedia *("our oracle of modern times")*," Jan. 2001. http://en.wikipedia.org.

[3] M. Bücker and P. Hovland, "Autodiff.org - Community Portal on Automatic Differentiation," 2000. http://www.autodiff.org.

[4] A. Walther, Andrea; Kowarz and A. Griewank, *ADOL-C User's Guide*, July 2005. http://www.math.tu-dresden.de/~adol-c/adolc110.ps.

[5] T. Neckel and T. Weinzierl, "Peano *("just another fluid solver")*." http://www5.in.tum.de/forschung/CFD/doxys/howto/index.html.

[6] S. Balay, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "PETSc User's Manual," Tech. Rep. ANL-95/11 - Revision 2.3.3, Argonne National Laboratory, 2007. http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf.

[7] S. Turek and M. Schäfer, "Benchmark computations of laminar flow around a cylinder," in *Flow Simulation with High-Performance Computers II* (E. H. Hirschel, ed.), no. 52 in NNFM, Vieweg, 1996.

[8] A. Kowarz and A. Walther, "Tapeless forward differentiation in adol-c," Nov. 2004. http://www.cranfield.ac.uk/dcmt/amsc/pdfs%20of%20talks/kowarznov04.pdf.