Bachelor Thesis

Comparison of Time Stepping Techniques for Compressible Gas Dynamics

Julian Köllermeier
RWTH Aachen University

Supervisor: Prof. Dr. Manuel Torrilhon
Center for Computational Engineering Science
Mathematics Division
RWTH Aachen
Abstract

This work presents an implicit software framework for the solution of non-linear partial differential equations. We use different time discretizations together with adaptive time stepping and error estimation. The emerging non-linear system of equations is solved by either a dual time stepping method or Newton’s method. For the solution of the large linear system of equations we employ the restarted GMRES and BiCGstab method combined with different preconditioners. As the computation of the Jacobian of the discretization is important, we compare different analytical and numerical approaches for the derivatives.

We investigate the effect of different parameter choices on the convergence behavior and runtime of the solution process. In the end, we obtain a substantial speedup resulting from the application of the methods mentioned above that shows the advantage of implicit methods for fluid dynamic simulations.
# Contents

<table>
<thead>
<tr>
<th>Contents</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xi</td>
</tr>
<tr>
<td>List of Symbols</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Numerical Simulations ................................. 1  
1.2 Aims of the project .................................. 1

## 2 Physical Background

2.1 Heat Conduction ........................................ 3  
2.2 Exact Solution of linear Heat Equation with Right Hand Side .............. 4  
2.3 Potential Flow ........................................ 5  
2.4 Navier-Stokes Equations ............................... 5

## 3 Numerics

3.1 Operator Splitting ..................................... 7  
3.2 Spatial Discretization .................................. 8  
3.3 Time Discretization ................................... 10  
3.3.1 Explicit and Implicit Schemes ....................... 10  
3.3.2 Runge-Kutta Schemes ................................ 11  
3.3.3 Consistency Conditions ............................ 12  
3.3.4 Stability ............................................ 13  
3.3.5 Implicit Euler Method ............................... 14  
3.3.6 Implicit Midpoint Rule ............................. 14  
3.3.7 Implicit Trapezoidal Rule ......................... 15  
3.3.8 Backward Difference Formula ..................... 15  
3.4 Adaptive Time Stepping ................................ 16  
3.4.1 Local and Global Error ............................ 17
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2</td>
<td>Error estimation</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Comparison with Different Order Method</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>Step Size Variation</td>
<td>19</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Richardson Extrapolation</td>
<td>20</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Time Step Control</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>General Control Theory Model</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>Integral Controller</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>PI Controller</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>PC Controller</td>
<td>25</td>
</tr>
<tr>
<td>3.5</td>
<td>Solution of Non-linear System</td>
<td>26</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Newton Algorithm</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>Dual Time Stepping</td>
<td>28</td>
</tr>
<tr>
<td>3.6</td>
<td>Calculation of the Jacobian</td>
<td>30</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Analytical Calculation</td>
<td>30</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Finite Differences</td>
<td>32</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Efficient Finite Differences</td>
<td>33</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Jacobian Free Method</td>
<td>35</td>
</tr>
<tr>
<td>3.7</td>
<td>Solution of Linear System</td>
<td>36</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Iterative Solvers</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>GMRES</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>BiCGSTAB</td>
<td>39</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Preconditioning</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td>ILU</td>
<td>41</td>
</tr>
<tr>
<td></td>
<td>SSOR</td>
<td>41</td>
</tr>
<tr>
<td>3.8</td>
<td>Compressed Row Storage</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>Analysis</td>
<td>45</td>
</tr>
<tr>
<td>4.1</td>
<td>Stability Analysis</td>
<td>45</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Model Problem</td>
<td>45</td>
</tr>
<tr>
<td>4.1.2</td>
<td>Analysis and Eigenvalue Computation</td>
<td>47</td>
</tr>
<tr>
<td>4.1.3</td>
<td>Stability Constraint</td>
<td>49</td>
</tr>
<tr>
<td>5</td>
<td>Results</td>
<td>51</td>
</tr>
<tr>
<td>5.1</td>
<td>Linear Solver</td>
<td>51</td>
</tr>
<tr>
<td>5.1.1</td>
<td>SSOR Preconditioner</td>
<td>52</td>
</tr>
<tr>
<td>5.1.2</td>
<td>GMRES</td>
<td>53</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Linear Solver Comparison</td>
<td>55</td>
</tr>
<tr>
<td>5.2</td>
<td>Non-linear Solver</td>
<td>58</td>
</tr>
<tr>
<td>5.3</td>
<td>Computation of the Jacobian</td>
<td>60</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Accuracy of Finite Differences</td>
<td>60</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Runtime of Different Approaches</td>
<td>61</td>
</tr>
<tr>
<td>5.4</td>
<td>Time Discretization</td>
<td>62</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Linear Test</td>
<td>62</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Non-linear Test</td>
<td>63</td>
</tr>
</tbody>
</table>
# Table of contents

5.5 Adaptive Time Stepping ........................................ 65  
5.6 Comparison with Explicit Method ............................ 68  
5.7 Spatial Discretization ......................................... 71  

6 Conclusion .......................................................... 73  

A Annex ............................................................... 75  

References ............................................................ 77
List of Figures

3.1 Stencil for the computation of $\kappa$ using left and right neighbor of face $i$. . 9
3.2 Feedback loop model of control theory . . . . . . . . . . . . . . . . . . . . 22
3.3 Stencil for the computation of the value at the cell center . . . . . . . . . . 30
3.4 Stencil for the computation of the fluxes over a horizontal face. . . . . . . . 31
3.5 Stencil for the computation of the fluxes over a vertical face. . . . . . . . . . 31
3.6 Example coloring of a $7 \times 6$ grid . . . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.1 maximum relaxation parameter $\alpha$ with respect to $\Delta \tau$ . . . . . . . . . . . . . . 50
5.1 SSOR iterations with respect to $\omega$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.2 preconditioned GMRES iterations with respect to $\omega$. . . . . . . . . . . . . . . . . . 53
5.3 Iterations and time with respect to restarts of GMRES . . . . . . . . . . . . . . . . . 54
5.4 Iterations and time for linear solvers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.5 Residual development for linear solvers . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.6 Residual development for DTS and Newton’s method. . . . . . . . . . . . . . . . . . 59
5.7 Relative error of FD with respect to $\epsilon$. . . . . . . . . . . . . . . . . . . . . . . . . . . 61
5.8 Relative error for different time discretizations (linear test). . . . . . . . . . . . . . . . . 63
5.9 Relative error for different time discretizations (non-linear test) . . . . . . . . . . . . 64
5.10 Step size development with respect to safety factor . . . . . . . . . . . . . . . . . . . 66
5.11 Step size development for different controllers . . . . . . . . . . . . . . . . . . . . . . . 67
5.12 Runtime of explicit versus implicit methods . . . . . . . . . . . . . . . . . . . . . . . . . . 70
List of Tables

3.1 Consistency conditions for Runge-Kutta methods .................................................. 13

5.1 $\omega_{opt}$ for SSOR ......................................................................................... 53
5.2 $\omega_{opt}$ for preconditioned GMRES ................................................................. 54
5.3 Optimal number of restarts for GMRES .............................................................. 55
5.4 Runtime in msec for different Jacobian calculations ............................................. 62
5.5 Timesteps and rejected steps for different controllers ........................................... 67
5.6 Runtime for different solution techniques ............................................................ 69
5.7 Relative error of spatial discretization ................................................................. 71
List of Symbols

Acronyms

- BDF: Backwards Difference Formula
- BiCG: Biconjugate Gradient Method
- BiCGSTAB: Biconjugate Gradient Stabilized Method
- CFL: Courant-Friedrichs-Lewy Number
- DTS: Dual Time Stepping
- FD: Finite Differences
- GMRES: Generalized Minimal Residual Method
- NSE: Navier-Stokes Equations
- PDE: Partial Differential Equation

Sub- and Superscripts

- $i$: face index
- $k$: dual time or Newton iteration index
- $n$: physical time iteration index

Symbols

- $T$: temperature
- $t$: physical time
- $\nabla$: gradient of a function
- $\kappa$: heat conductivity
- $\Delta$: Laplacian of a function
- $\eta$: viscosity
- $\lambda$: thermal conductivity
- $Q$: vector of flow variables
- $\tau$: dual time
- $\omega$: overrelaxation parameter
- $Kr_k$: Krylov subspace of dimension $k$
- $\epsilon$: step size for FD
Chapter 1

Introduction

1.1 Numerical Simulations

Modern engineering applications are more and more influenced by numerical simulations. During the past decades the development of efficient numerical codes together with major advances in computer architectures made it possible to simulate more complex models with a very high resolution.

The popular field of Computational Fluid Dynamics has provided many dedicated algorithms to solve very specific problems as well as more general types of equations. Though computer simulations are very powerful, they usually go hand in hand with experimental research because the calculated results need to be verified using real experiments. This is crucial because many of the numerical concepts rely on mathematical properties that do normally not describe the entire physical characteristics of the problem. After verification and validation of the simulation, the calculations can be extended to other problems or parameter regions were no experiments are available to obtain completely new results.

In order to do so, researchers all over the world develop general algorithms that can be used in a general framework afterwards to solve many different problems. In this context there is a large interest in the investigation of already existing algorithms under different circumstances. This directly brings us to the aims of this thesis.

1.2 Aims of the project

This project builds up on an existing software framework which was developed by KAPPER (see [10]). It was designed for simulations of different physical models using various explicit solution techniques and has been successfully tested before. The aim is now to implement an efficient implicit solution scheme for the simulation of fluid dynamic problems, especially for diffusion-type equations.

While trying to keep the framework as general as possible to allow user-friendly addition of new physical models, we want to employ different numerical methods to come up with an efficient solution of the emerging equation system.
The concrete implementation should include an implicit numerical scheme which is
time accurate and uses a dual time stepping scheme (DTS) or Newton’s method for the
non-linear equations. For the solution of the emerging large linear system of equations we
want to make use of different preconditioned linear solvers like GMRES and BiCGstab.
In a parameter study we want to find out optimal values for the parameters of the
different methods to ensure a good convergence behavior. For the computation of the
Jacobian matrix, we want to use analytical as well as numerical methods and compare
accuracy and runtime of the methods.

In the end, we want to make a comparison between the implicit methods and a basic
explicit solver to show the advantage of implicit methods.
Chapter 2

Physical Background

In general, we want to cover diffusion-type equations. But with the help of the operator splitting technique (see 3.1) it is also possible to simulate more arbitrary models. Therefore we will not only describe elliptic equations but also the well-known Navier-Stokes equations which contain convective terms.

Our general equation looks like

\[ \frac{\partial Q}{\partial t} + L(Q) = 0 \]  

(2.1)

With an elliptic operator \( L \). To see applications of this kind of equation we start with a equation.

2.1 Heat Conduction

We consider the two-dimensional heat equation on a non-regular grid. Therefore the problem equation reads

\[ \frac{\partial T}{\partial t} + \nabla (q) = 0 \]  

(2.2)

The heat flux \( q \) itself is modelled by Fourier’s law, saying that the heat flux through a surface is directly proportional to the negative temperature gradient across the surface (for more information see [9]).

\[ q = -\kappa(T)\nabla T \]  

(2.3)

Here \( \kappa \) is the thermal conductivity of the material and its value may depend on the value of the temperature \( T \) itself.

For example, we consider two possible models

\[ \kappa \equiv \text{const} \]  

(2.4)

\[ \kappa = \kappa(T) \]  

(2.5)

Model (2.4) will lead to a linear PDE and model (2.5) results in a non-linear equation, which has a large influence on the computations.
2.2 Exact Solution of linear Heat Equation with Right Hand Side

In order to allow for error calculations of our different methods mentioned in Chapter 3, we are interested in an exact solution of a simple test case. We therefore consider the linear heat equation with \( \kappa = \lambda \equiv const \) on a square domain \( \Omega = [0, 1] \times [0, 1] \). We set homogeneous boundary conditions \( T(t, x) = 0 \) at \( x = 0 \) and \( x = 1 \) and we set Neumann boundary conditions at \( y = 0 \) and \( y = 1 \) resulting in a quasi one-dimensional setting. We start from initial condition \( T(0, x) = 0 \). As for the right hand side of our equation, we take a simple sine function \( f(x) = 10 \cdot \sin(\pi x) \) so that the equation to solve is

\[
\frac{\partial T}{\partial t} + \lambda \frac{\partial^2 T}{\partial x^2} = f(x) \quad (2.6)
\]

It is possible to solve this equation analytically with the help of a special ansatz

\[
T(t, x) = \sum_{k=1}^{\infty} \alpha_k(t) \cdot \sin(k\pi x) \quad (2.7)
\]

We furthermore develop the right hand side in a general Fourier series

\[
f(x) = \sum_{k=1}^{\infty} \beta_k \cdot \sin(k\pi x) \quad (2.8)
\]

Computing the corresponding derivatives and inserting the ansatz as well as the Fourier series into Equation 2.6 yields

\[
\sum_{k=1}^{\infty} \left( \frac{\partial \alpha_k(t)}{\partial t} + \lambda k^2 \pi^2 \alpha_k(t) - \beta_k \right) \cdot \sin(k\pi x) = 0 \quad \forall k \in \mathbb{N} \quad (2.9)
\]

As this has to hold for every \( x \in \Omega \), the first factor of every term has to be zero, thus

\[
\frac{\partial \alpha_k(t)}{\partial t} + \lambda k^2 \pi^2 \alpha_k(t) - \beta_k = 0 \quad \forall k \in \mathbb{N} \quad (2.10)
\]

This is a set of ordinary differential equations for the coefficients \( \alpha_k(t) \) and the solution can be easily derived as

\[
\alpha_k(t) = c_k e^{-\lambda k^2 \pi^2 t} + \frac{\beta_k}{\lambda k^2 \pi^2} \quad (2.11)
\]

The coefficient \( c_k \) can be obtained with the help of the initial condition \( \alpha_k(t = 0) = 0 \). This leads to

\[
c_k = -\frac{\beta_k}{\lambda k^2 \pi^2} \quad (2.12)
\]

Inserting the resulting \( \alpha_k(t) \) into the ansatz gives us the solution of Equation 2.6

\[
T(t, x) = \sum_{k=1}^{\infty} \frac{\beta_k}{\lambda k^2 \pi^2} \left( 1 - e^{-\lambda k^2 \pi^2 t} \right) \cdot \sin(k\pi x) \quad (2.13)
\]
Choosing \( f(x) = 10 \cdot \sin(\pi x) \), the fourier coefficients \( \beta_k \) are easy to obtain, in fact we have

\[
\beta_k = 0 \quad \forall \, k \neq 1, \quad \beta_1 = 10 \tag{2.14}
\]

The solution then simplifies to

\[
T(t, x) = \frac{1}{\lambda 10 \pi^2} \left( 1 - e^{-\lambda 100 \pi^2 t} \right) \cdot \sin(10 \pi x) \tag{2.15}
\]

We see, that the solution approaches a stationary sine wave and the influence of the initial condition decays exponentially. The calculation of the exact solution now allows for comparison with numerical solutions to compare the componentwise error or respective norms (see Chapter 5).

### 2.3 Potential Flow

The potential equation is a special case of the stationary heat equation. For example, it can describe the inviscid flow past an airfoil. Assuming irrotational flow, i.e. \( \overrightarrow{\omega} = \text{rot} \overrightarrow{v} = 0 \), the momentum equations are already fulfilled. Furthermore, the potential \( \Phi \) is introduced as

\[
\begin{align*}
    u &= \frac{\partial \Phi}{\partial x} & v &= \frac{\partial \Phi}{\partial y}
\end{align*} \tag{2.16}
\]

Which is consistent with the assumption \( \omega = 0 \)

Continuity equation \( \text{div} \overrightarrow{v} = 0 \) then gives us the Laplace equation or potential equation as follows

\[
\Delta \Phi = 0 \tag{2.17}
\]

Whereas the velocity of the flow field can be later derived using \(2.16\). More details about potential flows can be found in [26].

As the equation is linear, there should not be any problems concerning stability and therefore this equation is very good for a first test of any solver.

The implementation presented in this project is treating time depending PDEs. This means that the solution of the potential equation is obtained by approaching the steady state of the general time dependent equation.

### 2.4 Navier-Stokes Equations

The Navier-Stokes equations describe the general motion of fluids. In contrast to the potential flow, viscous terms are included making the system of equations very difficult to solve. The conservation of mass, momentum and energy in two dimensions can be
written as follows
\[ \frac{\partial}{\partial t} \rho + \frac{\partial}{\partial x} (\rho u) + \frac{\partial}{\partial y} (\rho v) = 0 \] (2.18)
\[ \frac{\partial}{\partial t} (\rho u) + \frac{\partial}{\partial x} (\rho u^2 + p + \tau_{xx}) + \frac{\partial}{\partial y} (\rho u v + \tau_{xy}) = 0 \] (2.19)
\[ \frac{\partial}{\partial t} (\rho v) + \frac{\partial}{\partial x} (\rho u v + \tau_{xy}) + \frac{\partial}{\partial y} (\rho v^2 + p + \tau_{yy}) = 0 \] (2.20)
\[ \frac{\partial}{\partial t} (\rho E) + \frac{\partial}{\partial x} (\rho u E + u \tau_{xx} + v \tau_{xy} + q_x) + \frac{\partial}{\partial y} (\rho v E + v p + v \tau_{yy} + u \tau_{xy} + q_y) = 0 \] (2.21)

Here \( \tau \) is the symmetric stress tensor, which can be modelled for a Newtonian fluid as
\[ \tau_{xx} = -\eta \left( 2 \frac{\partial u}{\partial x} - \frac{2}{3} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right) \] (2.22)
\[ \tau_{xy} = -\eta \left( \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \] (2.23)
\[ \tau_{yy} = -\eta \left( 2 \frac{\partial v}{\partial y} - \frac{2}{3} \left( \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right) \] (2.24)

And the heat fluxes are given by Fourier’s law again
\[ q_x = -\kappa \frac{\partial T}{\partial x} \] (2.25)
\[ q_y = -\kappa \frac{\partial T}{\partial y} \] (2.26)

With temperature \( T \) computed from an equation of state, e.g. \( T = T(E, p) \). \( \eta \) is called the viscosity and \( \kappa \) is again the thermal or heat conductivity, which can be assumed constant for the most applications. Nevertheless non-linear models for the stresses and the parameters can also be easily implemented.

For a derivation and a more detailed explanation about the Navier-Stokes equations, we suggest again.
Chapter 3

Numerics

After modeling the physical processes, we end up with a set of equations. The aim of the numerics is now to translate these equations into numbers. In the first Section 3.1 we will give a guide of how to deal with convective terms in the equations. This typically requires a discretization of both the time and the spatial derivatives which is explained in Sections 3.3 and 3.2. Every discretization comes up with an error in the solution and the aim is of course to control this error. In order to choose the physical time step of the time discretization as large as possible with respect to a certain error threshold, we will also talk about adaptive time stepping in Section 3.4. Here we explain the topic from a control theory approach and also give some examples for practical use. Nevertheless, in the end one has to solve a non-linear system of equations for which we want to use either a dual time stepping algorithm or Newton’s method as explained in Section 3.5. Both these methods have in common that they linearize the non-linear system, so we will also give an overview about the different possibilities of computing the Jacobian of the equation system in Section 3.6. In the last Section 3.7 we provide the last ingredient for the simulation, namely the iterative linear solvers and preconditioners.

3.1 Operator Splitting

Many physical models can be written in the following form

$$\frac{\partial}{\partial t} Q + \sum_i \frac{\partial}{\partial x_i} F_{x_i}(Q) = L(Q) \quad (3.1)$$

Where $Q$ is the vector of the unknown variables (e.g. $Q = T$ in case of the heat equation), $F_{x_i}(Q)$ is the convective flux vector in the respective direction (e.g. $F_{x_i}(Q) = 0$ for the heat equation as there is no convection) and $L(Q)$ is a diffusive term or source term on the right hand side of the equation (e.g. $L(Q) = \triangle T$ for the heat equation with constant $\kappa = 1$). Note that also the diffusive term are usually given by corresponding fluxes, like for the heat equation.

The fluxes of the Navier-Stokes equations (2.18) can also be split into diffusive and convective terms. The diffusive terms include all the terms of the stress tensor and
the heat fluxes, whereas the convective terms are given by combinations of velocity and pressure.

This distinction is made because the physical processes of convection and diffusion are very different and therefore require special numerical treatments. A convective term for example is modeling transport in a certain direction so that an appropriate upwinding scheme is the method of choice for the numerical solution. In contrast to that, transport phenomena modeled by diffusive terms describe the spreading of particles, mass or energy without any distinct direction. This is the reason why a symmetric stencil should be used for the discretization of the operators or the reconstruction of the diffusive fluxes.

In order to separate the treatment of those two different effects, one can split the original PDE into two systems as follows

\[
\begin{align*}
\frac{\partial}{\partial t} Q_1 + \text{div} F(Q_1) &= 0 \\
\frac{\partial}{\partial t} Q_2 &= L(Q_2)
\end{align*}
\]

Now the single processes can be treated separately, but in order to get the solution of the original system \((3.1)\) one has to somehow put the solutions of the single systems together.

This can be carried out by using the computed solution of the first equation as an initial condition for the next step of the second equation and vice versa. The scheme then consists of two steps on each time level.

First the convection equation is solved for one step using the old solution from the diffusion equation as initial condition

\[
\begin{align*}
PDE & : \frac{\partial}{\partial t} \overline{Q} + \frac{\partial}{\partial x} F(\overline{Q}) = 0, \\
IC & : \overline{Q}(x, t^n) = Q^n
\end{align*}
\]

Then the diffusion equation is solved for one step using the old solution from the convection equation as initial condition

\[
\begin{align*}
PDE & : \frac{\partial}{\partial t} Q = L(Q), \\
IC & : Q(x, t^n) = \overline{Q}^{n+1}
\end{align*}
\]

The fashion can be extended to a second order scheme, where a full step of the diffusive solution is preceded and followed by two half steps of the convective solution. For more details, the reader is referred to [23].

### 3.2 Spatial Discretization

The spatial discretization is very important for the implementation of the method, as it will be necessary to evaluate the Jacobian of the discretized fluxes. So we first have to explain the spatial discretization.
We employ an approach by Kapper, who uses a six-point stencil for the reconstruction of the fluxes over the cell boundaries including a least squares solution. We will here only describe the procedure for the two-dimensional heat equation. The application of the same ideas can then easily be adopted for the other models.

With the help of Gauss’ integral theorem, the differential operators are first reformulated into fluxes over the four cell boundaries so that the fluxes become

\[ \sum_{i=0}^{4} F_{i,x}(Q) \cdot n_{i,x} + F_{i,y}(Q) \cdot n_{i,y} \]  

(3.2)

Where \( F_{i,x} \) and \( F_{i,y} \) are the fluxes over cell boundary \( i \) in \( x \) and \( y \) direction, which are in case of the heat equation

\[ F_{i,x}(T) = \kappa_{i}(T) \frac{\partial T}{\partial x_{i}}, \quad F_{i,y} = \kappa_{i}(T) \frac{\partial T}{\partial y_{i}} \]  

(3.3)

The heat conductivity is averaged using the values of \( \kappa \) from the neighbor cells like in figure 3.1

\[ \kappa_{i}(T) = \frac{1}{2} (\kappa(T_{L}) + \kappa(T_{R})) \]  

(3.4)

Note that this choice is different from an approach, where the temperature is first averaged as follows

\[ \kappa_{i}(T) = \kappa \left( \frac{1}{2} (T_{L} + T_{R}) \right) \]  

(3.5)

For the evaluation of the derivatives of \( T \) or \( Q \) in general, we consider a linear reconstruction of the values in the following way

\[ Q(x, y)_{i} = a_{i,0} + a_{i,1} \cdot x + a_{i,2} \cdot y \]  

(3.6)

Now the derivatives are easy to obtain

\[ \frac{\partial Q}{\partial x_{i}} = a_{i,1}, \quad \frac{\partial Q}{\partial y_{i}} = a_{i,2} \]  

(3.7)

It is important to know, that the coefficients \( a_{i,j} \) are calculated using a least-squares approach which includes the six surrounding cells. The least-squares problem imposes

![Figure 3.1: Stencil for the computation of \( \kappa \) using left and right neighbor of face \( i \)](image-url)
six conditions on the reconstruction $Q$ thus leading to an over determined system for
the three variables $a_{i,j}$. These parameters depend on the surrounding six values of
$T$ according to each face’s stencil. Considering $M$ as the matrix of the least-squares
problem we get

$$
\begin{pmatrix}
a_{i,0} \\
a_{i,1} \\
a_{i,2}
\end{pmatrix} = M^{-1} \cdot \begin{pmatrix} T_{i,0} \\ \vdots \\ T_{i,5} \end{pmatrix}
$$

(3.8)

For the boundary cells, the fluxes are the same, but the coefficients are calculated
using

$$
\begin{pmatrix}
a_{i,0} \\
a_{i,1} \\
a_{i,2}
\end{pmatrix} = \begin{pmatrix}
a_{i,0} \\
a_{i,1} \\
a_{i,2}
\end{pmatrix} - N_i \cdot \begin{pmatrix} a_{i,0} \\ a_{i,1} \\ a_{i,2} \end{pmatrix} - D_i
$$

(3.9)

The additional matrices $N_i, C_i, D_i$ contain information about the boundary types
for example. They modify the constraints of the least squares problem such that the
boundary conditions are satisfied. For additional details, we recommend [10]).

3.3 Time Discretization

After the spatial discretization of our general equation 2.1 we obtain the following
equation

$$
\frac{\partial Q}{\partial t} + R(Q) = 0
$$

(3.10)

where we have a time derivative of the flow variable $Q$ and the result of the spatial
discretization of the elliptic operator $R(Q)$.

In order to calculate the flow variable we now need to discretize the physical time
derivative term. This means that we replace the first term by an approximation of the
derivative. This step is very important for the whole simulation as the discretization
method has major influence on the stability properties and the accuracy as well as the
effort needed for the simulation.

3.3.1 Explicit and Implicit Schemes

Having performed the time discretization of Equation 2.1 we will end up with a non-
linear equation like

$$
g(Q^{n+1}, Q^n) = 0
$$

(3.11)

Where the the function $g$ also includes evaluations of the operator $R$. In most cases,
the equation can be rewritten into the following form

$$
Q^{n+1} = Q^n + f(Q^{n+1}, Q^n)
$$

(3.12)

Which takes care of the effect that the old value of the flow variable $Q^n$ is updated
and then results in the new value $Q^{n+1}$. 
In general, one distinguishes between explicit and implicit methods. In explicit methods, the update itself only depends on the value of the old flow variable $Q^n$, whereas the update for the implicit methods also depends on the new flow variable $Q^{n+1}$. Summarized, that means

$$Q^{n+1} = Q^n + f(Q^n) \quad \text{explicit method} \quad (3.13)$$

$$Q^{n+1} = Q^n + f(Q^{n+1}, Q^n) \quad \text{implicit method} \quad (3.14)$$

Obviously, an explicit method allows for a very fast calculation of the new value $Q^{n+1}$ as it only needs to calculate the right hand side terms which only include old values. But the drawback of these methods is, that they are only stable up to a usually very small physical time step $\Delta t$. Thus one has to do many time steps in order to calculate the solution at a later time. The size of the time step is closely related to the so called CFL number, which is a function of a ratio of the time step size and the smallest grid cell length $CFL = f \left( \frac{\Delta t}{\Delta x} \right)$. For explicit methods, this number usually includes model parameters like the heat conductivity and the CFL number has to be smaller than a certain constant value, e.g. 1, for stability reasons. Thus, one has to choose a very small time step for very fine grids, too. In stability theory this is due to the fact, that the numerical domain of dependence has to include the physical domain of dependence. As the new flow variable only depends on old values of the surrounding cells, the numerical domain of dependence is finite and therefore restricts the physical time step size.

On the other hand, for the implicit methods it is typically not so easy to calculate the update, as this includes the solution of a non-linear system of equations in every single step. This increases the effort per time step dramatically. But the big advantage of implicit methods is, that they are in general unconditionally stable with respect to the linear theory. So the stability of the method does not depend on the time step chosen. As a consequence, the method allows for much larger time step sizes than the explicit method. For an implicit method, the new solution at a single point depends on all the other values of the old time step, so the numerical domain of dependence includes the whole physical domain and thus there is no time step restriction.

Throughout this whole work, we want to cover implicit schemes, as the time step constraints for our applications have shown to be too restrictive to allow for efficient explicit computations.

### 3.3.2 Runge-Kutta Schemes

After the spatial discretization, we end up with a large coupled system of ordinary differential equations (ODEs). A general framework for the solution of ODEs is given by the Runge-Kutta schemes (see [6] for a detailed description about Runge-Kutta schemes and the topics discussed in this Section). We therefore consider a generalization of Equation 2.1.

Given an initial value problem

$$y'(t) = f(t, y(t)); y(0) = y_0, y : \mathbb{R} \rightarrow \mathbb{R}^n$$

(3.15)
an $s$-stage Runge-Kutta method calculates the new solution as follows

$$y_{n+1} = y_n + h \cdot \sum_{j=1}^{s} b_j k_j$$  \hspace{1cm} (3.16)$$

with time step size $h = t_{n+1} - t_n$. The coefficients $b_j$ can be seen as the coefficients of a quadrature formula for $\int_{t_n}^{t_{n+1}} f(t, y(t)) dt$. The intermediate values $k_j$ are evaluations of the right hand side function $f$:

$$k_j = f(t_n + hc_j, y_n + h \cdot \sum_{l=1}^{s} a_{jl} k_l), \quad j = 1, \ldots, s$$  \hspace{1cm} (3.17)$$

Where we can think of $c_j$ and $a_{ji}$ being coefficients of a quadrature formula for the intermediate values $k_j$.

Obviously the method becomes explicit if $a_{jl} = 0$ for all $l \geq j$. Otherwise, it is implicit and one has to solve a system of equations to compute the $k_j$ in order to get $y_{n+1}$.

The different coefficients are usually arranged in the so called Butcher tableau (after John C. Butcher, \[2\]) as follows:

$$
\begin{array}{c|cccc}
 c & a_{11} & a_{12} & \cdots & a_{1s} \\
 c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\
 \vdots & \vdots & \vdots & \ddots & \vdots \\
 c_s & a_{s1} & a_{s2} & \cdots & a_{ss} \\
 \hline
 & b_1 & b_2 & \cdots & b_s \\
\end{array}
$$  \hspace{1cm} (3.18)$$

Note that the method is explicit, if the matrix $A$ is a strict lower triangular matrix, otherwise it is implicit.

**Consistency Conditions**

A time stepping method is called consistent of order $p$, if

$$\frac{1}{h} \| y(t_n + h) - y_{n+1} \| = O(h^p)$$  \hspace{1cm} (3.19)$$

meaning that the error of the simulation, so the difference of the exact solution $y(t_n + h)$ starting with $y(t_n) = y_n$ and the approximation $y_{n+1}$ has to decrease in the specified way.

The conditions for the different consistency orders are easy to derive by means of Taylor expansion and summarized in the following table (see \[6\] for higher order conditions).

It is also important that the highest possible consistency order $p$ never exceeds the number of stages $s$: $p \leq s$ for explicit Runge-Kutta methods. Implicit methods can still have higher consistency orders as we will see.
\[
p = 1 \quad \sum_i b_i = 1 \\
p = 2 \quad \sum_i b_i c_i = \frac{1}{2} \\
p = 3 \quad \sum_i b_i c_i^2 = \frac{1}{3} \text{ and } \sum_{ij} b_i a_{ij} c_i = \frac{1}{6}
\]

Table 3.1: Consistency conditions for Runge-Kutta methods

Stability

A more rigorous approach to verify the stability of an integration method (like any Runge-Kutta method) than the physical explanation from Section 3.3.1 is to apply it to the one-dimensional linear model equation

\[y'(t) = \lambda y, \quad y(0) = 1 \quad (3.20)\]

By direct substitution we end up with an iteration of the form

\[y_{n+1} = R(h\lambda)y_n \quad (3.21)\]

The behavior of the iteration now depends on the so called stability function \(R(z)\) as every perturbation during the solution process is multiplied by the stability function. We directly see, that \(|R(z)| < 1\) leads to a reduction of errors while \(|R(z)| > 1\) amplifies the perturbations. Hence, we define the stability region \(S\) of the method

\[S := \{z \in \mathbb{C} : |R(z)| \leq 1\} \quad (3.22)\]

The integration method then gives stable results if \(h\lambda \in S\).

Furthermore the method is A-stable if the stability region includes the complete left half plane of the complex plane. The property therefore leads to a stable behavior of the solution for all negative values of \(\lambda\).

A different stability property is L-stability. A schemes is called L-stable, if it is A-stable and if its stability function satisfies the following condition

\[\lim_{\text{Re}(z) \to -\infty} |R(z)| = 0 \quad (3.23)\]

This property leads to a fast damping of oscillations during the iterations.

As an example, the stability function for the explicit Euler method

\[y_{n+1} = y_n + h \cdot f(t_n, y_n) \quad (3.24)\]

looks as follows

\[R(z) = 1 + z \quad (3.25)\]

Resulting in a quite small stability region which is in fact a circle in the left half of the complex plane.

\[S = \{z = x + iy : (1 + x)^2 + y^2 < 1, x, y \in \mathbb{R}\} \quad (3.26)\]
For a value of \( \lambda = -1 \), for example, the largest possible value for the time step is \( h = 2 \).

In contrast, the implicit Euler method (see next Section \[3.3.2\]) comes up with a very different stability function

\[
R(z) = \frac{1}{1 + z}
\]  

(3.27)

Which means that the stability region includes the whole left half of the complex plane. Therefore the method is A-stable, resulting in a monotonically decreasing sequence \( y_n \) for every negative \( \lambda \). Thus, the method is stable for every negative value of \( \lambda \). This is the great advantage of implicit methods in general. There exist more definitions of related stability concepts such as \( A(\alpha) \)-stability, see [7] for more detailed information.

**Implicit Euler Method**

The simplest implicit time stepping method to solve the general initial value problem \[3.15\] is the implicit Euler method, sometimes also called the backward Euler method as it can be motivated using backward differences for the time derivative on the left hand side. The method is defined as follow

\[
y_{n+1} = y_n + h \cdot f(t_n + h, y_{n+1})
\]  

(3.28)

Translating this to the Butcher tableau yields

\[
\begin{array}{c|c}
1 & 1 \\
1 & 1
\end{array}
\]  

(3.29)

We can see, that the implicit Euler method is a single stage, implicit Runge-Kutta scheme. The method is of consistency and convergence order \( p = 1 \) and like all implicit methods unconditionally stable. Furthermore we obtain A-stability and L-stability by checking the definitions from Section \[3.3.2\].

**Implicit Midpoint Rule**

A slight difference to the previous method can be made to obtain the implicit midpoint rule. As the name already states, the integration of the initial value problem \[3.15\] is done using the midpoint rule as follows

\[
y_{n+1} = y_n + h \cdot f \left( t_n + \frac{h}{2}, \frac{y_{n+1} + y_n}{2} \right)
\]  

(3.30)

The only difference to the implicit Euler method is the averaging of the argument of the right hand side function.

In the Butcher tableau the method can be written as

\[
\begin{array}{c|c}
\frac{1}{2} & 1 \\
\frac{1}{2} & 1
\end{array}
\]  

(3.31)
It can be seen from the consistency order conditions 3.1 that this method is of consistency order $p = 2$ and again unconditionally stable.

The stability function can be written as

$$R(z) = \frac{1 + \frac{1}{2}z}{1 - \frac{1}{2}z}$$  \hspace{1cm} (3.32)

which gives us A-stability but not L-stability meaning that though the method is stable, it can have some undesired oscillations. We will see this property again in the results Chapter 5.

**Implicit Trapezoidal Rule**

As single stage Runge-Kutta methods are sometimes not accurate enough, it is important to use multi stage methods like the implicit trapezoidal rule. Unlike the methods before, this method needs two function evaluations.

$$y_{n+1} = y_n + h \cdot \frac{1}{2} \cdot (f(t_n + h, y_{n+1}) + f(t_n, y_n))$$  \hspace{1cm} (3.33)

The difference to the implicit midpoint rule is, that now the function evaluations are averaged and not the argument itself. Thus, for a linear function $f(y) = A \cdot y$ with a matrix $A$ the methods give the same results.

The Butcher tableau now also has two stages

\[
\begin{array}{c|ccc}
0 & 0 & 0 \\
1 & \frac{1}{2} & \frac{1}{2} \\
\frac{1}{2} & 1 & 1
\end{array}
\]  \hspace{1cm} (3.34)

The order conditions are satisfied up to $p = 2$, so that the method is of consistency order 2.

As the stability analysis is performed for the linear model equation 3.20 and the trapezoidal rule reduces to the implicit midpoint rule in the linear case, we end up with the same stability properties as the implicit midpoint rule. Hence, we have A-stability but not L-stability.

**3.3.3 Backward Difference Formula**

Besides the well-known Runge-Kutta schemes, there is a second family of methods, that is widely used to solve initial value Problems like Equation 3.15. The so called *backwards difference formulas* (BDF) approximate the derivative on the left hand side using previously computed values. Therefore they can be seen as linear multistep methods

$$\sum_{i=0}^{k} a_i y_{n+1-i} = h \sum_{i=0}^{k} b_i f(y_{n+1-i})$$  \hspace{1cm} (3.35)

with the time step size $h$ and a set of coefficients $a_i, b_i$. 
For the original form of a general k-stage BDF we set \( b_i = 0 \) for \( i > 0 \) and write the method as
\[
\sum_{i=0}^{k} \alpha_i y_{n+1-i} = hf(y_{n+1}) \tag{3.36}
\]
where \( \alpha_i := \frac{a_i}{b_i} \).

As the right hand side term \( f(y_{n+1}) \) contains the unknown new variable \( y_{n+1} \), this method is also implicit and needs the solution of a non-linear system of equations.

When starting the simulation, the values \( y_j \) for \( j < 0 \) are usually not available and must be obtained by some kind of extrapolation from the initial condition or by other lower order methods.

We will now give a short overview of the most important BDF methods.

**Implicit Euler Method**

It is very easy to obtain the simple implicit Euler method. This can be done setting \( k = 1, \alpha_0 = 1, \alpha_1 = -1 \). Then we get the method already described in Section 3.3.2
\[
y_{n+1} - y_n = hf(y_{n+1}) \tag{3.37}
\]

**BDF2**

Another real multistep method is the so called BDF2 method. It uses the two last values to compute the value corresponding to the new time level and the coefficients are chosen such that the method is of order \( p = 2 \), which can be easily verified by Taylor expansion.

Approximating the left hand side derivative of Equation 3.15 by one sided second order finite differences in \( y \), we obtain the coefficients for the BDF2 method as \( k = 2, \alpha_0 = \frac{3}{2}, \alpha_1 = -2, \alpha_2 = \frac{1}{2} \).

This results in the general formulation
\[
\frac{3}{2} y_{n+1} - 2y_n + \frac{1}{2} y_{n-1} = hf(y_{n+1}) \tag{3.38}
\]

### 3.4 Adaptive Time Stepping

The accuracy of the solution highly depends on the time step size \( h \) as the consistency definition 3.19 shows. Therefore we can expect a good approximation for small values of \( h \) and a rather coarse approximation for large values of \( h \). But in some regions of smooth behavior it might be sufficient to take large timesteps and still get a good accuracy. In order to understand this, we first need to introduce the definitions of local and global errors according to [23].
3.4.1 Local and Global Error

The local error $l_e_n$ is defined as follows

$$l_e_n = u(t_n + h) - y_{n+1}$$

(3.39)

where $u(t_n + h)$ is the exact solution calculated from initial value $u(t_n) = y_n$ and $y_{n+1}$ is the numerical solution obtained by a Runge-Kutta method for example. Note that $e_n$ is only the local error introduced in the current time step and not directly connected to the global error of the whole time interval of interest.

The global error is defined as

$$g_e_n = y(t_n) - y$$

(3.40)

and here $y(t_n)$ is the exact solution calculated from the initial value $y(t_0) = y_0$. $y_n$ is again the numerical solution of the equation at time $t_n$. The difference is, that the global error takes into account the accumulation of local errors throughout the calculation. Thus, the error propagation is important. For a stable method, error damping occurs and therefore

$$||y(t_{n+1}) - y_{n+1}|| \ll ||g_e_n||$$

(3.41)

When accumulating the local errors of the whole time interval of the simulation we obtain the expansion of the approximate solution $y_{n+1}$ as follows (compare [4])

$$y_{n+1} = y(t_{n+1}) + \Phi(t_n)h^p + \mathcal{O}(h^{p+1})$$

(3.42)

In contrast to that, one can easily show by Taylor expansion that a method of order $p$ has a local error of

$$e_n = \Phi(t_n, y_n)h^{p+1} + \mathcal{O}(h^{p+2})$$

(3.43)

where $\Phi$ is called the principal error function. This principal error function usually depends on the derivative of the right hand side function $f$. In smooth regions, the derivative of $f$ is rather small so the local error might be small, regardless the size of the time step $h$. Therefore it is indeed possible to choose a larger time step and end up with a small error. On the other hand, when the function $f$ is highly non-linear and the value of $y$ changes rapidly, the step size should be reduced, because a high value of the principle error function might lead to a large error instead.

In order to adjust the time step to this different behavior of the principal error function, we need two different ingredients:

1. an estimate of the error
2. a strategy to adjust the physical time step

We will cover the two topics in the next sections.
3.4.2 Error estimation

For an efficient adaptive time step control it is necessary to estimate the error after every step of the numerical algorithm. As described before, there are two available definitions of the error, the global and the local error.

A quantity $\epsilon$ is called error estimate for the unknown error $\epsilon$, if there exist two constants $c_1, c_2$ with $c_1 \leq 1 \leq c_2$ such that

$$c_1 \epsilon \leq \bar{\epsilon} \leq c_2 \epsilon$$

(3.44)

Which means, that the error estimate is neither over- nor underestimating the error too much.

It is also important for the error estimate to be exact for vanishing step size $h$. We say an error estimate $\bar{\epsilon}$ is asymptotically exact if it satisfies

$$\lim_{h \to 0} ||\bar{\epsilon}|| = ||\epsilon||$$

(3.45)

Comparison with Different Order Method

One possibility to estimate the local error of the time stepping method is to compare the solution to a better approximation of higher order as described by [23]. Therefore, we need two time stepping methods. The first method which is assumed to be of order $p$ produces the approximate solution $\hat{y}_{n+1}$. The second method is more accurate and so the solution $y_{n+1}$ approximates the real solution with order $p + 1$. Now the local errors of the first and second method are defined according to 3.39 as

$$\hat{e}_n = u(t_n + h) - \hat{y}_{n+1}$$

(3.46)

$$e_n = u(t_n + h) - y_{n+1}$$

(3.47)

As an error estimate for the local error of the first (coarse) method, we now take the difference of the two solutions:

$$\hat{\epsilon} = y_{n+1} - \hat{y}_{n+1}$$

(3.48)

$$\epsilon = y_{n+1} - u(t_n + h) + u(t_n + h) - \hat{y}_{n+1}$$

(3.49)

$$\hat{e}_n - e_n$$

(3.50)

Or respectively

$$||\hat{\epsilon}|| = ||\hat{e}_n - e_n||$$

(3.51)

Using

$$0 \leq \theta := \left|\left|\frac{e_n}{\hat{e}_n}\right|\right| \leq 1$$

(3.52)

the definition 3.44 is satisfied because of the triangle inequality

$$(1 - \theta)||\hat{e}_n|| = \left(1 - \left|\left|\frac{e_n}{\hat{e}_n}\right|\right|\right)||\hat{e}_n|| = ||\hat{e}_n|| - ||e_n||$$

(3.53)

$$||\hat{e}_n|| - ||e_n|| \leq ||\hat{e}_n - e_n||$$

(3.54)

$$||\hat{\epsilon}||$$

(3.55)
and
\[
\|\hat{e}\| = \|\hat{e}_n - e_n\| \leq \|\hat{e}_n\| + \|e_n\| \quad (3.56)
\]
\[
= (1 + \|\frac{e_n}{\hat{e}_n}\|)\|\hat{e}_n\| \quad (3.57)
\]
\[
= (1 + \theta)\|\hat{e}_n\| \quad (3.58)
\]

As the two methods have order \(p\) and \(p+1\) respectively, it holds
\[
\|e_n\| = O(h^{p+2}) \quad (3.59)
\]
\[
\|\hat{e}_n\| = O(h^{p+1}) \quad (3.60)
\]

Such that the asymptotic stability is satisfied because of
\[
\theta = \|\frac{e_n}{\hat{e}_n}\| \to 0 \quad \text{for } h \to 0 \quad (3.61)
\]

Though this property is desirable, it has the drawback that it estimates the error of the first method which is not as accurate as the second method. Consequently, any time step control mechanism would adjust the time step such that the error of the inaccurate method is controlled. This is somewhat unsatisfactory because one would rather like to use the accurate method for the calculation of approximate solutions. The solution to this problem is to assume that the adjusted time step will also lead to a good control of the error of the accurate method (see also \([4]\)). It is a very simple trick and still reasonable as the error of the accurate method should in any case be smaller than the error of the coarse method.

**Step Size Variation**

A different approach is to estimate the global error of the method. Therefore, we make use of the expansion of the approximate solution (see Equation 3.42) for the same method with different time step size \(h\). A summary of this method can also be found in \([24]\).

Using a method of order \(p\) and time step size \(h\) we get
\[
y_n^h = y(t_{n+1}) + \Phi(t_n)h^p + O(h^{p+1}) \quad (3.62)
\]

In contrast, we arrive at the following relation for the same method using a time step size of \(\frac{h}{m}\) for \(m \in \mathbb{N}\)
\[
y_n^m = y(t_{n+1}) + \Phi(t_n)\frac{h}{m}^p + O\left(\frac{h^{p+1}}{m}\right) \quad (3.63)
\]

Neglecting higher order terms and computing the difference, we obtain
\[
y_n^h - y_n^m = \Phi(t_n)(1 - m^{-p})h^p \quad (3.64)
\]
This equation can be solved for the unknown principal error function $\Phi$

$$\Phi(t_n) = \frac{y_{n+1}^h - y_{n+1}^m}{(1 - m^{-p})h^p} \quad (3.65)$$

We can now estimate the global error of one of the methods using the estimate for the principal error function, e.g. for the coarse method

$$y_{n+1}^h - y(t_{n+1}) \approx \frac{y_{n+1}^h - y_{n+1}^m}{(1 - m^{-p})h^p} \quad (3.66)$$

and for the fine method respectively

$$\frac{h}{m} y_{n+1}^m - y(t_{n+1}) \approx \frac{y_{n+1}^h - y_{n+1}^m}{(1 - m^{-p})h^p \frac{m}{m}} = \frac{y_{n+1}^h - y_{n+1}^m}{(m^p - 1)} \quad (3.67)$$

### 3.4.3 Richardson Extrapolation

A very useful benefit of the step size variation method is, that it directly estimates the value of the principal error function $\Phi$. Due to that we can insert the estimate in Equation 3.42 and solve for the (unknown) real solution. Due to that, we get a higher order approximation of the solution. This approach was first proclaimed by Richardson in 1911 (see [15] for the original article).

$$y(t_{n+1}) = \frac{h}{m} y_{n+1}^m - \Phi(t_n) \frac{h^p}{m} + O\left(\frac{h^{p+1}}{m}\right) \quad (3.68)$$

$$= \frac{h}{m} y_{n+1}^m - \frac{y_{n+1}^h - y_{n+1}^m}{(1 - m^{-p})h^p \frac{m}{m}} + O\left(\frac{h^{p+1}}{m}\right) \quad (3.69)$$

$$= \frac{m^p \cdot y_{n+1}^m - y_{n+1}^h}{m^p - 1} + O\left(\frac{h^{p+1}}{m}\right) \quad (3.70)$$

So by redefining the numerical solution we gain one order of accuracy compared to the original numerical solution:

$$\tilde{y}_{n+1} := \frac{m^p \cdot \frac{h}{m} y_{n+1}^m - y_{n+1}^h}{m^p - 1} \quad (3.71)$$

This procedure could be extended to even higher order gains by contribution of more fractional step sizes. For example, one could think of three methods with step sizes $h, \frac{h}{2}, \text{ and } \frac{h}{4}$ to end up with an error of the order of $h^{p+2}$.

The question of now, why should one try to reduce the error using the extrapolation method rather than using smaller time step sizes as predicted by any kind of time step control? The answer is that for very small step sizes, which could be necessary to reduce the error of a fixed order method to a given tolerance, the round off errors have to
be taken into account. Due to very small numbers, cancelation errors lead to a higher numerical error than the method’s own discretization error. There are test cases, for which the error of a method only scales with the time step for time step sizes up to $10^{-4}$, for example, meaning that smaller time step sizes conversely lead to larger errors (see [22] for an example). So the only possibility is to reduce the error either by choosing another (higher order) time stepping technique or local extrapolation techniques as described above. As higher order methods are not always at hand and may lead to a very different behavior of the solution, the Richardson extrapolation might often be a good choice. It is especially handy as it needs only few additional calculations to update the original value. Furthermore, it can be used in a black-box manner, where the user only has to provide values for the order of the method and the number of fractional steps for the error estimate (one often chooses $n = 2$ for a simple and cheap comparison).

3.4.4 Time Step Control

The general aim of an adaptive time stepping method is to control the time step of the numerical method such that the error (or the error estimate) is bounded by a certain threshold and the method therefore generates accurate results while calculating with a rather large time step value. This can also be seen as a control theory problem. In control theory, the aim is to control the output of a process through the adjustments of an input parameter. In our case, the output is the error of the method and the input parameter is then the time step size $h$.

As control theory provides many good possibilities to deal with the problem mentioned above, we will now take a closer look at the general philosophy of control theory before we give some examples for practical use of special controllers to choose the time step size adaptively.

General Control Theory Model

In general, control theory assumes an arbitrary process which output (in our case the error of the method) is controlled by a change in one of the input parameters of that process (in our case the time step size $h$). The general control theory model is the so-called feedback loop explained in Figure 3.2.

As we can see, the system generates a certain output. This can be seen as the numerical method that results in an error after every single time step. This error is then measured by some kind of sensor, for example one of the error estimators above. The measured output error is compared with a reference value for the error, for example a user given threshold $\text{tol}$. The difference is called the measured error, referring to the difference between the desired and the given output value. The controller takes this as an input value and transforms it into a signal which is then given back to the process again. In our case, this system input would be the time step size $h$.

We always have a disturbance that influences the system. One could think of round-off errors or any other effects that cannot easily be modeled. It is the task of the controller to take care of these disturbances. As we expect the method’s error to be
related to the time step size, we hope to change the error in the desired way such that there is no longer any difference to the reference value.

It is very important, that the feedback loop does not lead to an unstable behavior of the closed loop dynamics. Therefore we need to adjust the controller to the system that is used. Not every controller is applicable for every system, but there are some controllers which are very suitable for time step control in particular.

As our system only produces output values after every single iteration step, we have a discrete process and thus also a discrete controller operating on discrete input values.

It should be said, that the behavior of the error estimator is very important for the success of the time step control. To be more precise, the error must be observable and controllable. The first means that the estimate of the error mirrors the real relation between the step size and the error. Furthermore, the system is controllable, if it is able to effectively change the error output through variations of the step size input. [20] gives a more detailed overview about the different properties like observability and controllability in general.

Before we explain how the process is modeled and which controllers can be used, we first need to introduce the notation. We refer to [20] for a more detailed description of the notation.

We take the error estimate $\hat{\epsilon}$ from Equation (3.51) and define the error per step (EPS) as

$$\hat{r}_{n+1} = ||\hat{\epsilon}_{n+1}||$$

In case we want to control the error per unit step (EPUS) we define respectively

$$\hat{r}_{n+1} = \frac{||\hat{\epsilon}_{n+1}/h_n||}{h_n}$$

We want to achieve $\hat{r}_{n+1} \leq \text{TOL}$. For practical use, it is better to aim at a slightly smaller error $\epsilon = \theta \cdot \text{TOL}$, where $\theta$ can be interpreted as a safety factor.

A first example of a controller is the elementary local error control algorithm de-

---

**Figure 3.2: Feedback loop model of control theory**
scribed in [20]

\[ h_{n+1} = \left( \frac{\epsilon}{\hat{r}_{n+1}} \right)^{\frac{1}{k}} h_n \quad \text{(3.74)} \]

with \( k = p + 1 \) for EPS and \( k = p \) in case of EPUS.

In order to understand the choice of this simple controller, we take a look at the underlying heuristics with the help of the following

Process assumptions

1. Asymptotics: \( \hat{r}_{n+1} = \hat{\phi}_n h^k_n \) where \( \hat{\phi}_n = ||\hat{\Phi}_n|| \) and \( \hat{\Phi}_n \) is an estimate for the principal error function at step \( n \)

2. Slow variation: \( \hat{\phi}_n \approx \hat{\phi}_{n-1} \)

The first relation essentially demands, that \( h_n \) is so small, that the asymptotic behavior is already valid. The second assumption requires a smooth behavior of the right hand side function which changes the value of the principle error function. Nevertheless, these assumptions do not always hold in practice.

Inserting these two relations into our elementary control equation 3.74 we directly obtain (compare to [20])

\[ h_{n+1} = \left( \frac{\epsilon}{\hat{\phi}_n h^k_n} \right)^{\frac{1}{k}} h_n \quad \text{(3.75)} \]

which leads to

\[ \epsilon = \hat{\phi}_n h^k_{n+1} \quad \text{(3.76)} \]

Such that we exactly meet the error tolerance that we aimed for. This is also called deadbeat control as it instantly removes the error.

The elementary controller therefore seems to make sense. Of course we want to search for better methods. The general control theory framework provides various methods for this purpose. We will now take a closer look at some of the most useful controllers for adaptive time stepping.

At this point we like to emphasize that the following stability comments only rely on the asymptotics assumption but not on the assumption of slow variation.

**Integral Controller**

In fact, the elementary controller 3.74 is know as an integral controller (I controller) in control theory (see [20] [21]).

To clarify the name, we look at the logarithm of Equation 3.74 and first obtain the difference equation

\[ \log h_{n+1} = \log h_n + \frac{1}{k} (\log \epsilon - \log \hat{r}_{n+1}) \quad \text{(3.77)} \]

According to [20] \( \log \epsilon - \log \hat{r}_{n+1} \) is called the control error, while the factor \( k_I = \frac{1}{k} \) is the integral gain.
By recursion, we arrive at the discrete form of an integral with starting value \( \log h_0 \) and therefore refer to this method as *integral control*.

\[
\log h_{n+1} = \log h_0 + k_I \sum_{m=1}^{n} (\log \epsilon - \log \hat{r}_n) \tag{3.78}
\]

Together with the process which is essentially modeled by the asymptotic relation \( \hat{r}_{n+1} = \hat{\phi}_n h_n^k \) or \( \log \hat{r}_{n+1} = \log \hat{\phi}_n + k \log h_n \) we obtain the so called *closed loop dynamics*

\[
\log h_{n+1} = (1 - kk_I)\log h_n + k_I (\log \epsilon - \log \hat{\phi}_n) \tag{3.79}
\]

For \( kk_I = 1 \) or \( k_I = \frac{1}{k} \) we get back to our original elementary controller, but we can now also change the value of \( k_I \) (note that \( k \) is in general fixed by the time stepping method). The reason why we should do so is, that \( \log h_{n+1} \) is proportional to \( \log \hat{\phi}_n \) for \( kk_I = 1 \). Thus, the step size sequence \( h_n \) shows the same smoothness behavior as the principal error function in that case. This can lead to oscillatory behavior at the edge of stability of the time stepping method. In order to avoid this, we may choose different values for \( kk_I \).

The characteristic equation of the controller (3.78) is

\[
q = 1 - kk_I \tag{3.80}
\]

and the roots of this equation determine the stability region of the closed loop process (negative roots are required to damp errors). Therefore, we have to choose \( kk_I \in [0, 2] \). The different behavior of the method can be summarized as follows

- \( kk_I \in [0, 2] \) : stability \hspace{1cm} (3.81)
- \( kk_I \in (1, 2] \) : fast but oscillating behavior \hspace{1cm} (3.82)
- \( kk_I = 1 \) : elementary control, deadbeat \hspace{1cm} (3.83)
- \( kk_I \in (0, 1) \) : slow and smooth control \hspace{1cm} (3.84)

We do not want to go into the details of the stability calculations here, for more detailed information the reader is referred to [20].

Again we write down the integral controller

\[
h_{n+1} = \left( \frac{\epsilon}{\hat{r}_n} \right)^{k_I} h_n \tag{3.85}
\]

to underline the fact, that the parameter \( k_I \) is not fixed by the method. It can be chosen such that the method exhibits the desired behavior.

**PI Controller**

As we have seen before, the integral controller accumulates the control errors of the previous steps to construct the control \( \log h_n \). An enhanced version of this controller is the
PI controller, where the PI denotes the *proportional-integral* behavior of the controller. In contrast to the controller before, we just add a term proportional to the control error to the integral term as follows

\[
\log h_{n+1} = \log h_0 + k_l \sum_{m=1}^{n} \left( \log \epsilon - \log \hat{r}_m \right) + k_P (\log \epsilon - \log \hat{r}_n) \tag{3.86}
\]

The parameter \( k_P \) is called *proportional gain* and has to be chosen together with \( k_I \) to get a good behavior of the closed loop dynamics (see [21] for more information on how to choose the parameters).

Similar to the approach before, we form a recursion to end up with the equation

\[
\log h_{n+1} = \log h_n + k_I (\log \epsilon - \log \hat{r}_{n+1}) + k_P (\log \hat{r}_n - \log \hat{r}_{n+1}) \tag{3.87}
\]

which can be transformed into the usual form for the calculation of the new time step size (compare to Equation 3.85)

\[
h_{n+1} = \left( \frac{\epsilon}{\hat{r}_{n+1}} \right)^{k_I} \left( \frac{\hat{r}_n}{\hat{r}_{n+1}} \right)^{k_P} h_n \tag{3.88}
\]

From Equation 3.88 we directly see the influence of the proportional term. If the error estimate \( \hat{r} \) is increasing, the new time step is smaller (of course, this only holds for positive proportional gains \( k_P \)). On the other hand, the step size increases faster, if the error estimate reduces.

The characteristic equation of this controller is now a second order polynomial with two roots. We will not go into detail for the choice of the right parameters as this also requires some knowledge about numerical stability limits (see [20] for more information).

A pair of frequently used parameters is \((k k_I, k k_P) = (0.3, 0.4)\). Together with a safety factor of \( \theta = 0.8 \), we arrive at the so called PI.3.4 controller

\[
h_{n+1} = \left( 0.8 \cdot \text{TOL} \right)^{0.3/k} \left( \frac{\hat{r}_n}{\hat{r}_{n+1}} \right)^{0.4/k} h_n \tag{3.89}
\]

**PC Controller**

A more advanced method for choosing the stepsize is a *predictive controller*. The prediction can be done by the construction of an observer for the principle error function \( \phi \). The formula for the new time step size can be obtained according to [20]

\[
h_{n+1} = \left( \frac{\epsilon}{\hat{r}_{n+1}} \right)^{k_E} \left( \frac{\hat{r}_n}{\hat{r}_{n+1}} \right)^{k_R} \frac{h_n}{h_{n-1}} \tag{3.90}
\]

This controller is similar to a PI controller for the difference \( \log(h_{n+1}) - \log(h_n) \) hopefully leading to a smoother step size sequence if the parameters are chosen appropriately.

In [20], the author suggests \( k_E = 0.4 \) and \( k_R = 0.7 \) for a smooth step size development.
3.5 Solution of Non-linear System

After performing a spatial and time discretization as described in Sections 3.2 and 3.3 one ends up with a large system of equations. The equations are in general non-linear and cannot be easily solved for the unknown variables $Q^n$. Hence, we need to employ an efficient non-linear solver to calculate the solution at every physical time step. In the following, we describe the dual time stepping and the Newton algorithm to deal with this problem. Again, the general form of the equation we are talking about will be (compare with 2.1)

$$\frac{\partial Q}{\partial t} + R(Q) = 0 \quad (3.91)$$

And the time derivative is discretized depending on one of the specific time discretizations mentioned in Section 3.3 above leading to

$$\frac{Q^{n+1} - Q^n}{\Delta t} + R(Q^{n+1}) = \tilde{R}(Q_{IE}^{n+1}) = 0 : \text{implicit Euler} \quad (3.92)$$

$$\frac{Q^{n+1} - Q^n}{\Delta t} + R\left(\frac{Q^{n+1} + Q^n}{2}\right) = \tilde{R}(Q_{IM}^{n+1}) = 0 : \text{implicit Midpoint} \quad (3.93)$$

$$\frac{Q^{n+1} - Q^n}{\Delta t} + \frac{1}{2}(R(Q^{n+1}) + R(Q^n)) = \tilde{R}(Q_{IT}^{n+1}) = 0 : \text{implicit Trapezoidal} \quad (3.94)$$

$$\frac{3Q^{n+1} - 4Q^n + Q^{n-1}}{2\Delta t} + R(Q^{n+1}) = \tilde{R}(Q_{BDF2}^{n+1}) = 0 : \text{BDF2} \quad (3.95)$$

In the following Sections, we will summarize the different time discretizations above using the formulation

$$\tilde{R}(Q^{n+1}) = 0 \quad (3.96)$$

3.5.1 Newton Algorithm

The Newton algorithm is a very well-known and easy to use method to solve non-linear systems of equations. It is in principle based on a simple Taylor expansion and then forms an iteration to converge to the real solution (a detailed investigation of Newton’s method can be found in [3]). First we need a starting value, say $Q^{n+1}_k = Q^n$. Starting from 3.96 we expand the left hand side to obtain

$$\tilde{R}(Q_{k+1}^{n+1}) \approx \tilde{R}(Q_{k+1}^n) + \frac{\partial \tilde{R}}{\partial Q}(Q_{k+1}^{n+1} - Q_{k+1}^n) \quad (3.97)$$

where we neglected higher order terms. As we hope for the new value $Q_{k+1}^{n+1}$ to be a good approximation to the root of Equation 3.96 we also assume $\tilde{R}(Q_{k+1}^{n+1}) \approx 0$ and arrive at the new linear equation

$$\frac{\partial \tilde{R}}{\partial Q}(Q_{k+1}^{n+1} - Q_{k+1}^n) = -\tilde{R}(Q_{k+1}^n) \quad (3.98)$$
Introducing the update \( \Delta Q_{k+1}^{n+1} = Q_{k+1}^{n+1} - Q_{k}^{n+1} \), we only have to solve one linear system and update the old value afterwards to obtain the new iterate as follows

\[
\begin{align*}
\frac{\partial \tilde{R}}{\partial Q} \Delta Q_{k+1}^{n+1} &= -\tilde{R}(Q_k^{n+1}) \quad (3.99) \\
Q_{k+1}^{n+1} &= Q_k^{n+1} + \Delta Q_{k+1}^{n+1} \quad (3.100)
\end{align*}
\]

By using the gradient \( \frac{\partial \tilde{R}}{\partial Q} \) to calculate the update, we approach in the direction of decreasing residual \( \tilde{R}(Q_{k+1}^{n+1}) \) which is the desired behavior. Nevertheless, it can happen that the residual actually increases due to a strong non-linearity of the function. In that case, the first order Taylor expansion in \( 3.97 \) is to inaccurate and thus leading to bad results. A possibility to remedy this is a so called line search in the direction of the update. Therefore we compute the update just as in Equation \( 3.99 \), but afterwards update the solution according to

\[
Q_{k+1}^{n+1} = Q_k^{n+1} + \alpha \Delta Q_{k+1}^{n+1} \quad (3.101)
\]

where the parameter \( \alpha \) is chosen such that the residual is in fact decreasing. This can be done by successive testing, i.e. one first tests \( \alpha = 1 \) which should be the default value.

If the residual is larger, one tests \( \alpha = 1/2 \) leading to a damping of the update and so on. Conversely, it is possible to increase the value of \( \alpha \) in smooth regions to accelerate convergence.

Of course, every adjustment step of \( \alpha \) includes the calculation of the residual \( \tilde{R}(Q_{k+1}^{n+1}) \) which in turn means that the whole physical time and spatial discretization has to be calculated. This typically requires a lot of effort and should not be done too often.

The use of the parameter \( \alpha \) can be treated as a kind of time step of the Newton algorithm, as it determines the size of the update and plays an important role for the accuracy as well as for the stability of the method. Hence, we can in principle also apply the adaptive time stepping techniques mentioned in Section \( 3.4 \) to \( \alpha \). The only difference is, that the Newton method does not need to be time accurate in terms of the time stepping methods covered in \( 3.3 \). In Section \( 4.1 \) we describe a possibility to estimate the largest possible value for \( \alpha \) that still ensures stability of the Newton method in our multi-block framework by means of an eigenvalue computation.

The Newton method shows a quadratic convergence. So if \( Q_{k+1}^{n+1} \) is the solution of the non-linear system \( 3.96 \) it exists a constant \( C \geq 0 \) such that (compare \( 3 \))

\[
||Q_{k+1}^{n+1} - Q_{**}^{n+1}|| \leq C||Q_{k}^{n+1} - Q_{**}^{n+1}||^2 \quad (3.102)
\]

But this result holds only locally which means that one cannot expect convergence for every initial value. In general, the initial value \( Q_{k=0}^{n+1} \) must be sufficiently close to the real solution \( Q_{**}^{n+1} \). For some functions (like \( f(x) = \sqrt{|x|} \) in the scalar case) it is even possible that the original method is diverging for every initial guess as the residual is increasing in every step. The line search described above is one method to overcome this problem but it can lead to a very small \( \alpha \) reducing the rate of convergence dramatically.
So in case of very large physical time steps $\Delta t$ we expect a large difference between $Q^{n+1}_*$ and the initial value $Q^n$ so that the Newton algorithm in its simple formulation might fail. In that case, one should not change the physical time step size $\Delta t$ but the parameter $\alpha$ as described before.

One disadvantage that cannot be avoided is that the algorithm uses the derivative (or often called the Jacobian) of the function $\tilde{R}$. In practical applications it is infeasible to calculate the exact derivative for every underlying flow equation as well as the special spatial and time discretizations. In fact is is more common to approximate the derivative by some kind of difference formula. One then speaks of inexact or quasi Newton methods as the approximative Jacobian might differ from the exact derivative. The calculation of an approximative Jacobian can be very costly and requires some extra considerations. See Section 3.6 for different approaches of this derivation.

A very useful advantage of Newton’s method is its rapid convergence for linear equations. The reason is that the higher order terms in the Taylor expansion vanish for linear functions $\tilde{R}(Q) = AQ, A \in \mathbb{R}^{n \times n}$ because of $\frac{\partial^n \tilde{R}}{\partial Q^n} = 0$ for $n > 1$. Thus Equation 3.97 is exact and the iteration converges after one step.

### 3.5.2 Dual Time Stepping

In order to speed up convergence of unsteady flows, the dual time stepping method (first described by Merkle in [14]) can be used. It introduces a new artificial time derivative into the discretized partial differential equation 3.96. For example, we may consider

$$\frac{\partial Q}{\partial \tau} + \tilde{R}(Q^{n+1}) = 0 \quad (3.103)$$

The time $t$ is called the physical time, whereas the pseudo or artificial time $\tau$ is called the dual time.

Now there are several possibilities to discretize the dual derivatives. As already explained, an explicit method has a certain time step constraint in order to be numerically stable. To overcome those problems, a fully implicit scheme can be used, which we describe in the following.

Hence, for the discretization of the dual time derivative, we use backwards difference formulas. Choosing a simple implicit Euler method leads to

$$\frac{Q^{n+1}_k - Q^n_k}{\Delta \tau} + \tilde{R}(Q^{n+1}_{k+1}) = 0 \quad (3.104)$$

Note that we can also derive methods for higher order in dual time using different Runge-Kutta schemes for example (see Section 3.3.2). But it is important to say that we are not interested in a time accurate solution with respect to the dual time. We only want to solve the linear system and therefore want to approach the steady state in the dual time as soon as possible.

In order to be able to solve for the unknown $Q^{n+1}_{k+1}$, we linearize the right hand side $-\tilde{R}(Q^{n+1}_{k+1})$ to get the update of the dual time step:
\[ \tilde{R}(Q_{n+1}^k) \approx \tilde{R}(Q_{n}^{k+1}) + \frac{\partial \tilde{R}}{\partial Q} \cdot \Delta Q_{k+1}^{n+1} \] (3.105)

\[ \Delta Q_{k+1}^{n+1} = Q_{n+1}^k - Q_k^{n+1} \] (3.106)

Note that the linearization does not introduce an error in the case of a linear equation (e.g. heat equation with constant heat conductivity) because the reconstruction of the fluxes is also linear.

Now we can write down the linear system of equations

\[ \frac{\Delta Q_{k+1}^{n+1}}{\Delta \tau} = - \left( \tilde{R}(Q_{n}^{n+1}) + \frac{\partial \tilde{R}}{\partial Q} \Delta Q_{k+1}^{n+1} \right) \] (3.107)

After rearranging terms, we get the equation that we have to solve for the update \( \Delta Q_{k+1}^{n+1} \)

\[ \left( \frac{1}{\Delta \tau} I + \frac{\partial \tilde{R}}{\partial Q} \right) \Delta Q_{k+1}^{n+1} = -\tilde{R}(Q_{n}^{n+1}) \] (3.108)

The value for the dependent variable \( Q \) at the new physical time step \( n+1 \) is now obtained by convergence of the inner dual time loop: \( Q_{k+1}^{n+1} \rightarrow Q_{n+1}^{n+1} \) as \( k \rightarrow \infty \)

Compared to Newton’s method above (see Section 3.5.1) we see a strong similarity. The only difference is the term \( \frac{1}{\Delta \tau} I \) that is added to the Jacobian on the left hand side. We obviously get the Newton method in the limit of large dual time steps \( \Delta \tau \rightarrow \infty \). The small extra term on the left hand side in fact improves the stability of the method as it increases the diagonal values of the matrix. If the diagonal values of the matrix are larger then the sum of the (usually negative) non-diagonal entries in the same row

\[ |a_{i,i}| > \sum_{i \neq j} a_{i,j}, \quad \forall i \] (3.109)

a matrix is called diagonally dominant. This property yields better conditioning numbers of the matrix and also improves the solvability of the system with respect to the linear solvers covered in Section 3.7. For symmetric matrices, the diagonal dominance leads to positive definiteness of the matrix. Such matrices are usually very well conditioned and allow for very efficient solutions of the corresponding linear systems. As can be seen, small values of \( \Delta \tau \) most probably translate into large diagonal values of the system matrix and thus stabilize the system. On the other hand, small dual time step sizes also slow down the convergence to the steady state as many more steps are required in that case. Depending on stability problems, we can now apply a dual time step control to calculate with larger dual time steps where possible. Similar to the Newton algorithm, we are not interested in a time accurate solution and our only constraint is therefore the stability of the method. In Section 4.1 we describe the influence of the dual time step size on the stability for multi-block calculations.
3.6 Calculation of the Jacobian

After the discretization, all the different terms have to be calculated in order to start solving the large system of equations. Equations 3.99 and 3.108 for example include the evaluation of the Jacobian. So one has to find a way to compute this Jacobian efficiently with respect to accuracy, memory storage and runtime. We will now explain different approaches to get the Jacobian. It is important to say, that each method has its own benefits but also certain drawbacks that may be important or not depending on different aspects which we will explain in detail.

3.6.1 Analytical Calculation

As an example of the computation of the Jacobian, we explain the calculation for the heat equation assuming nonlinear heat conductivity.

If we consider a linear equation and all the steps of the flux reconstruction are also linear, the term \( \frac{\partial \tilde{R}}{\partial T} \) will not depend on the values of the solution, (for example for the heat equation with constant heat conductivity 2.4). Because of that the calculation of the corresponding term can be carried out before the iteration procedure starts in this case. In the case of variable heat conductivity 2.5 the equation becomes non-linear and the Jacobian has to be evaluated after every change in the variable \( T \).

Obviously, a certain value only couples with other values inside the stencil of the discretization. As the fluxes are reconstructed using a six-point stencil, we get a nine-point stencil for the cell values. The numbering of the cells is chosen as follows

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3.3: Stencil for the computation of the value at the cell center

Now one has to get the derivative of the right hand side \( \tilde{R}(T) \) which is basically the derivative of the spatial discretization using the fluxes over the boundary plus one extra term for the physical time discretization.

The most difficult part here is to consider the values inside the flux stencil with the right indices for the horizontal and vertical faces.

Here also the derivative of the heat conductivity \( \kappa \) has to be taken into account.
Using the ansatz $\omega$ the derivative looks like

$$\frac{\partial \kappa}{\partial T} = \omega \cdot T^{\omega - 1} \quad (3.110)$$

The final computation of the coefficients for the system matrix in the notation $AT = b$ can be seen in appendix A as an example for the dual time stepping applied to the implicit Euler method.

For the application it is important to know that the matrices $M, N, C$ used in the computation of the coefficients are constant throughout the whole flow simulation for each cell. It is possible to calculate all the coefficients before the simulation starts and there is no recalculation necessary, if one uses the constant heat conductivity because then it holds $\partial \kappa_{i,j} = 0$ and so the Jacobian does no longer depend on the temperature itself.

As one can easily see in A the analytical derivation of the Jacobian is a very elaborate task. Not only the structure of the equations themselves, but also the discretization has to be taken into account. This leads to an error-prone exercise where small mistakes can cause totally wrong results afterwards. Furthermore, some complex reconstructions and discretizations might not even have an analytical derivative or the derivative is at least not easy to obtain. This is one of the main disadvantages of the analytical calculation of the Jacobian.

The obvious advantage is, that the solution is (except for numerical round-off errors) exact to machine precision. Due to the manual calculation, it is also easier to exploit...
the sparsity of the matrix efficiently.

### 3.6.2 Finite Differences

In spite of the benefits of an exact calculation of the Jacobian it can be sufficient to approximate the Jacobian with a certain accuracy. In order to do so one can use finite differences.

The finite difference usually approximates the derivative of a one-dimensional function \( f : \mathbb{R} \to \mathbb{R} \) due to the following relation

\[
\frac{\partial f}{\partial Q} = \frac{f(Q + \epsilon) - f(Q)}{\epsilon} + \mathcal{O}(\epsilon) \quad (3.111)
\]

So instead of the real slope of the function at the point of interest, one computes the slope of the secant through the specific point and another point on the graph of the function only a very small step size \( \epsilon \) away.

Of course, the result is not the exact derivative, but with decreasing step size \( \epsilon \), the error is reduced as the error is in principle proportional to \( \epsilon \).

For a multi-dimensional function \( f : \mathbb{R}^n \to \mathbb{R}^m \), the perturbation \( \epsilon \) has to point into the right direction in order to calculate the derivative with respect to a single variable.

For example, it is

\[
\frac{\partial f_i}{\partial Q_j} \approx \frac{f_i(Q + \epsilon \cdot e_j) - f_i(Q)}{\epsilon} \quad (3.112)
\]

Where \( e_j \) is the unit vector that has a one at the \( j \)-th entry.

To get the derivative of the whole function with respect to one variable, one might therefore calculate

\[
\frac{\partial f}{\partial Q_j} \approx \frac{f(Q + \epsilon \cdot e_j) - f(Q)}{\epsilon} \quad (3.113)
\]

Resulting in a vector of dimension \( m \) with the respective derivatives in every single entry.

To fill the whole Jacobian, the finite differences have to be calculated for every variable and then combined to a matrix of size \( m \times n \), whereas we usually have \( m = n \) as the system is quadratic. As every calculation needs two evaluations of the function \( f \) this means that in total \( 2 \cdot n \) would be needed. But as \( f(Q) \) is always the same for every derivative, this result can be stored and used for all single calculations. So the calculation of the Jacobian includes essentially \( n \) evaluations of the function \( f \) which is in our case the right hand side term \( \tilde{R} \) consisting of the discretization and the underlying reconstruction. It is easy to see, that the computational effort for this not acceptable with respect to runtime. For non-linear calculations, the whole procedure would have to be repeated after every dual or Newton iteration which increases the simulation time substantially.

Another drawback is the accuracy of the method. The calculation is highly sensitive to changes in the step size \( \epsilon \). Usually, small values give small discretization errors but can result in large numerical round-off errors due to the limited precision of the machine. On the other hand, large values do not suffer numerical round-off errors but yield large
discretization errors. So one has to find the optimal value $\epsilon_{\text{opt}}$ that is a trade-off between round-off errors and discretization errors.

There are several approaches in the literature to find an appropriate value for $\epsilon_{\text{opt}}$. A widely used ansatz is described in [11], where $\epsilon$ is set approximately to the square root of the machine accuracy. This is denoted as

$$
\epsilon = \sqrt{\epsilon_{\text{ps}}}
$$

One can also take into account the size of the current solution $Q$ with an appropriate norm, as follows

$$
\epsilon = \sqrt{\epsilon_{\text{ps}}} (\|Q\| + 1)
$$

This means, that for larger values of $Q$ a larger $\epsilon$ is chosen in order to avoid cancelation errors. A drawback of this method is, that the computation is more expensive and the size of $Q$ must be available.

In order to test the different approaches for $\epsilon$ we made tests for varying step sizes. The results are shown in Section 5.

With respect to memory consumption, there can be some problems because one does not always know a-priori which values of the Jacobian are non-zero. So there is still some work to exploit the sparsity of the matrix and store the values efficiently. It needs some insight into the equations to figure out, which equation depends on which variables. It is also important to know the stencil of the spatial discretization, because this essentially determines the number of non-zero entries per row of the matrix. Alternatively, one could use an existing sparse matrix format that is included in many basic linear algebra libraries. The data format we used is the compressed row storage format (CRS) described for example also in [1]. For details, see Section 3.8.

A big advantage of finite differences is that the considered function can be quite arbitrary. Perhaps the values of $\epsilon$ have to be adjusted but besides that one only needs a method to evaluate the function for a given value $Q$. This is especially useful for flow simulations as the right hand side has to be calculated in any case. So the function $f$ is already provided and the finite differences can be used in a black-box fashion to calculate the Jacobian.

### 3.6.3 Efficient Finite Differences

As mentioned before the Jacobian is usually sparse. This is due to the fact, that the solution at one point only depends on the values inside the stencil. The finite differences approach makes uses of this and changes a value to obtain a change in the surrounding values. But this requires many function evaluations and thus is very costly. In fact, only the non-zero values of the resulting finite difference $\left. \frac{df}{\partial q_j} \right|_{\epsilon=\epsilon_{\text{ps}}} \approx \frac{f(Q+\epsilon e_j)-f(Q)}{\epsilon}$ are of interest, but there are many zero entries that are still computed.

Instead of this, we can change the vector $Q$ in more than one point and calculate the derivative with respect to more points with only one function evaluation. Under the assumption that the points do not share any stencil points with each other, they
will lead to different non-zero entries inside their respective Jacobian columns. In other words the columns of the Jacobian do not have a non-zero in a common row. Then, this columns are called *structurally orthogonal* (see [5]). It is possible to compute a set of structurally orthogonal columns with only one evaluation of the function.

As an example, consider a $7 \times 6$ grid with our usual stencil 3.3. Now every point inside the stencil has to have another color (here indicated by different numbers). Furthermore, points with the same color must not be neighbors or be part of the same stencil anywhere in the grid. An example coloring is given in figure 3.6. Here we just take the numbers from the original stencil and proceed in the same fashion for every other nine-point block inside the grid.

Now the number of function evaluations is equal to the number of colors inside the color map. This means that for a one-dimensional equation (e.g. heat equation [2.2]) only 9 evaluations are needed. For a three-dimensional equation (e.g. NSE 2.18) this number increases to $27 = 9 \cdot 3$. Compared to the original finite difference approach which needs $n$ number of function evaluations where $n$ is the number of variables, this is still a large gain.

An important advantage of the efficient method is its grid size independence. No matter how many points the grid consists of, the number of function evaluations will always be the same. For the small example from figure 3.6, the standard approach needs 123 evaluations in case of the Navier-Stokes equations, but for a test application with more than 3200 points, this increases to more than 10000 evaluations which makes a huge difference.

This efficient finite difference method is designed to improve the runtime of the original finite difference method. It is commonly used nowadays and there also exist many algorithms for coloring the respective grid or equation system. Though the method has a much shorter runtime, it does not come up with worse accuracy than the standard finite differences as mentioned in 3.6.2. There are no larger round-off errors because the
computations are essentially the same. Therefore the method still depends on the value of \( \epsilon \) for which the same approaches as shown in the section before can be applied.

### 3.6.4 Jacobian Free Method

All the methods mentioned before calculate the values of the Jacobian explicitly, which means that the Jacobian is computed before the preconditioning or the evaluation of the matrix-vector product inside the linear solvers is performed. Another approach works completely without the explicit values of the Jacobian. The so-called Jacobian free \[11\] method directly evaluates the matrix-vector product of the Jacobian with another arbitrary vector.

The method is based upon the following mathematical relation of the matrix-vector product

\[
\frac{\partial f}{\partial Q} \cdot v \approx \frac{f(Q + \epsilon \cdot v) - f(Q)}{\epsilon}
\]  

Which can be verified using trivial Taylor expansion

\[
\frac{1}{\epsilon} (f(Q + \epsilon \cdot v) - f(Q)) \approx \frac{1}{\epsilon} \left( f(Q) + \frac{\partial f}{\partial Q} \cdot \epsilon v - f(Q) \right) = \frac{\partial f}{\partial Q} \cdot v
\]

For another example, we refer to \[12\].

Inside the linear solver (see 3.7) the Jacobian free approach can now easily be used for the calculation of the occurring matrix-vector products.

Similar to the finite differences approach, the choice of the right parameter \( \epsilon \) is very important for the accuracy of the approximation. In general, the same principles for the choice of \( \epsilon \) can be applied. Nevertheless, this leads directly to first disadvantages namely the value of \( \epsilon \) and the accuracy of the result. A further drawback is, that the Jacobian free method has to be called for every evaluation of the matrix-vector product and the result cannot be reused as the vector \( v \) is usually different. So assumed the result of \( f(Q) \) has been calculated before the Jacobian free method needs one additional evaluation of the function \( f \) for every matrix-vector product, which can be a lot. For example, for the GMRES\( (m) \) method with \( m = 20 \) and a maximum number of restarts of 30 there are \( 20 \times 30 = 600 \) possible calls of the Jacobian free method, for every solution of the linear system. Compared to a standard finite differences approach this might still be advantageous but it also depends on the convergence behavior of the linear solver.

The benefit of the method is, of course, that it does not need to store the values of the Jacobian explicitly. Due to that the overall memory consumption decreases, especially as the calculation of the Jacobian itself can possibly need a lot of temporal memory.

But there is also a problem related to the fact that the Jacobian entries are no longer stored. The basic preconditioning methods need to have access to those entries during their iterations. This applies to splitting based preconditioners as well as for example an incomplete LU decomposition. So one either has to use matrix-free methods itself for preconditioning or the Jacobian still has to be computed somehow in addition. The latest is often applied using a frozen Jacobian approach which means that the Jacobian
is not updated in every iteration. More information about this and the Jacobian free method in general can be found in [11].

3.7 Solution of Linear System

Because of the spatial discretization, neighbouring values couple and we get a large and sparse linear system of equations which has to be solved for the variable $T$

$$Ax = b$$ (3.118)

where we formally have in the case of Newton’s method

$$A := \frac{\partial \tilde{R}}{\partial Q} \quad \text{where } x := \Delta Q_{k+1}^{n+1} \quad \text{and } b := -\tilde{R}(Q_k^{n+1})$$ (3.119) (3.120) (3.121)

and in the case of the dual time stepping we end up with

$$A := \frac{1}{\Delta \tau} I + \frac{\partial \tilde{R}}{\partial Q} \quad \text{where } x := \Delta Q_{k+1}^{n+1} \quad \text{and } b := -\tilde{R}(Q_k^{n+1})$$ (3.122) (3.123) (3.124)

In order to solve the emerging system of equations, one should apply iterative methods, as it can be computationally too expensive to solve the whole system directly and exactly by means of inversion.

For this purpose, we use different preconditioned iterative linear solvers, which are applicable for every non-singular system matrix.

3.7.1 Iterative Solvers

Iterative solvers in general are different from direct solvers as they usually calculate an approximative solution to the linear system of equation and not the exact solution. Thus, the necessary number of iterations as well as the runtime depends on the convergence behavior while direct solvers give the exact solution after a predictable number of calculation steps. Iterative solvers are especially well suited for sparse systems as this reduces the computational cost of matrix operations needed inside the single iterations as we will see. We will only cover some widely used solvers, all the methods available can be found in [18], for example.
GMRES

In general, GMRES (generalized minimal residual method) is an iterative method for the numerical solution of large linear systems of equations like

$$Ax = b$$ (3.125)

One important advantage is, that there are no restrictions for the system matrix $A$. This is especially useful, as the matrix in our case can easily be non-symmetric depending on boundary conditions, grid and equation.

The GMRES method was developed by SAAD and SCHULZ in 1986 \[19\] and has been thoroughly used in many applications since then.

The method iteratively approaches the solution by finding a vector in the so called Krylov subspace with minimal residual. This Krylov subspace is denoted as

$$Kr_k = \text{span}\{r, Ar, A^2r, ..., A^{k-1}r\}$$ (3.126)

With the residual $r = Ax_n - b$. Instead of using the almost linear dependent vectors $r, Ar, ...$ for the computation, an orthogonal basis of the Krylov subspace is constructed. This is done using the Arnoldi iteration (see \[17\] for detailed explanations). For the update of the solution a least squares problem has to be solved

$$\min ||r_0||_{2e_1 - H_ky_k}$$ (3.127)

where $H_k$ is an upper Hessenberg matrix resulting from the Arnoldi process.

Afterwards, the update can be computed as

$$x_k = x_0 + Q_ky_k$$ (3.128)

where $Q_k = (v_1, ..., v_k)$ is the matrix composed of the orthogonal basis of $Kr_k$.

A possible problem for this particular method is, that the memory requirements are very high, as every iteration needs to store one additional vector of the size of the solution vector for the basis of the Krylov subspace.

In order to overcome this problem, one often uses a restarted GMRES method. This means, that only a few iterations are performed and then the calculation starts again with the last solution vector as start value. Due to that, the memory storage is limited. Considering $m$ as the number of maximum inner iterations before a restart, this method is often called the GMRES($m$) method.

In pseudocode, the method including left preconditioning with the matrix $P^{-1}$ (see Section 3.7.2) then looks like this

```
for $n = 0, 1, 2, ..., n_{\text{max}}$ do
    $r_n = P^{-1}(b - Ax_n)$ (apply preconditioner)
    $\gamma_0 := ||r_n||_2, v_1 := \frac{r_n}{\gamma_0}$
    for $j = 1, ..., m$ do
        $r_{n+1} = r_n - \gamma_j v_j$
        $\gamma_{j+1} := ||r_{n+1}||_2, v_{j+1} := \frac{r_{n+1}}{\gamma_{j+1}}$
        $y_{n+1} = Q_{n+1}v_{j+1}$
    end for
    $x_{n+1} = x_n + y_{n+1}$
end for
```
\(w := P^{-1}Av_j\) (apply preconditioner)

\[\text{for } i = 1, \ldots, k \text{ do}\]
\[h_{ij} := v_i^Tw\]
\[w := w - h_{ij}v_i\]
\[\text{end for}\]
\[h_{j+1,j} := \|w\|_2\]
\[\text{if } h_{j+1,j} = 0 \text{ then}\]
\[j:=m\]
\[\text{else}\]
\[v_{j+1} := \frac{w}{h_{j+1,j}}\]
\[\text{end if}\]
\[\text{solve LLS: } \min||r_0||_2 e_1 - H_ky_k||\]
\[\text{end for}\]
\[\text{update } x_{n+1} := x_n + Q_ky_k\]
\[\text{end for}\]

In contrast to that, using right preconditioning leads to a slightly changed algorithm

\[\text{for } n = 0, 1, 2, \ldots, n_{\text{max}} \text{ do}\]
\[r_n = b - Ax_n\]
\[\gamma_0 := \|r_n\|_2, v_1 := \frac{r_n}{\gamma_0}\]
\[\text{for } j = 1, \ldots, m \text{ do}\]
\[w := AP^{-1}v_j\] (apply preconditioner)
\[\text{for } i = 1, \ldots, k \text{ do}\]
\[h_{ij} := v_i^Tw\]
\[w := w - h_{ij}v_i\]
\[\text{end for}\]
\[h_{j+1,j} := \|w\|_2\]
\[\text{if } h_{j+1,j} = 0 \text{ then}\]
\[j:=m\]
\[\text{else}\]
\[v_{j+1} := \frac{w}{h_{j+1,j}}\]
\[\text{end if}\]
\[\text{solve LLS: } \min||r_0||_2 e_1 - H_ky_k||\]
\[\text{end for}\]
\[\text{update } x_{n+1} := x_n + Q_kP^{-1}y_k\] (apply preconditioner)
\[\text{end for}\]

Both algorithms rely on the calculation of an initial residual, which is here denoted by \(\gamma_0\). An important difference is that this residual is only in the preconditioned space in the case of left-preconditioning. As a consequence, a very small value of \(\gamma_0\) does
not necessarily imply a small residual of the equation that one wants to solve. On the other hand, right preconditioning calculates the exact residual. Of course, both variants converge to the same unique solution but the size of the residuals inside of the methods is not comparable.

In both cases there are two parameters, that ensure or control the convergence of the GMRES(m):

\[ m : \text{maximum dimension of the Krylov subspace} \quad (3.129) \]
\[ \epsilon_{ps} : \text{threshold for the residual} \quad (3.130) \]

In our calculations, we used for example \( m = 11 \) and \( \epsilon_{ps} = 1 \cdot 10^{-8} \) on a 40x40 grid and observed a good convergence behavior of the method. But these values might need to be changed in order to optimize the convergence speed for different physical or dual step sizes.

A value of \( 10^{-8} \) might seem not too small for a residual threshold of the GMRES(m) method, but it is not necessary to fully converge with the iterative solver as full convergence is not mandatory for inner dual or Newton iterations. So in order to save computational time, one should better do more non-linear iterations than resolve each single iteration more accurately. We observed that a value of \( \epsilon_{ps} = 10^{-8} \) is sufficient for the convergence of the dual iterations. Larger values can spoil the quality of the solution at every dual time step (or Newton step respectively) too much and therefore lead to a larger amount of iterations until convergence. Again, this also depends on the physical time step size.

**BiCGSTAB**

Another iterative method to solve the system 3.118 is the so-called biconjugate gradient stabilized method (BiCGSTAB, see [13] for a detailed description and derivation of the algorithm). This method can be motivated as a combination of the BiCG and the GMRES method where one GMRES(1) iteration follows each BiCG iteration leading to a much smoother residual behavior during the calculations. The BiCG algorithm itself constructs a basis of the transpose Krylov subspace

\[ K^T r_k = \text{span}\{r, A^T r, A^T 2r, ..., A^T (k-1)r\} \quad (3.131) \]

besides the basis of the normal Krylov subspace \( K r_k \)

The iterative solution of the BiCG method is then obtained by the Petrov-Galerkin orthogonality condition

\[ b - Ax_m \perp K^T m \quad (3.132) \]

In contrast to the BiCG method the BiCGSTAB method works without matrix-vector product of the transpose matrix \( A^T \). This is a big advantage as it does not require additional storage for the transpose or a special data format to store the matrix.

The algorithm given in the formulation of [13] looks as follows
\[ r_0 := p_0 := b - Ax_0, \quad \rho_0 := r_0^T r_0 \]

\begin{algorithm}
\begin{algorithmic}
\State \textbf{for} \( n = 0, 1, 2, \ldots, n_{\text{max}} \) \textbf{do}
\State \quad \( v_n := Ap_n, \quad \alpha_n := \frac{\rho_n}{v_n^T r_0} \)
\State \quad \( s_n := r_n - \alpha_n v_n, \quad t_n := As_n \)
\State \quad \( \omega_n := \frac{r_n^T s_n}{t_n^T t_n} \)
\State \quad \( x_{n+1} := x_n + \alpha p_j + \omega_n s_n \)
\State \quad \( r_{n+1} := s_n - \omega_n t_n \)
\State \quad \( \rho_{n+1} := r_{n+1}^T r_0, \quad \beta_n := \frac{\omega_n \rho_{n+1}}{\rho_n} \)
\State \quad \( p_{n+1} := r_{n+1} + \beta_n (p_n - \omega_n v_n) \)
\State \textbf{end for}
\end{algorithmic}
\end{algorithm}

The method can be stopped if either the maximum number of iterations \( n_{\text{max}} \) is achieved or the threshold \( r_{\text{min}} \) is reached by the residual \( r_n \), so that there are two parameters of the method.

Preconditioning can be easily done by applying BiCGSTAB to the preconditioned system.

A disadvantage in the algorithm is, that division with zero can occur even for regular matrices leading to a breakdown of the method without a solution. There are several approaches to overcome these problems for example using restarts similar to the GMRES method (see [13] for explanations).

Compared with the GMRES method above we see, that there are two matrix-vector multiplications during each iteration, while GMRES only needs one. Furthermore, we have a fixed number of four scalar products in each single BiCGSTAB iteration. For GMRES the number of necessary scalar products increases during the calculation as the dimension of the Krylov subspace grows. Therefore the computational effort of a single iteration is in average larger for the GMRES method. We refer to the result section 5 for simulation time comparisons between BiCGSTAB and the GMRES method.

3.7.2 Preconditioning

As a possibility to speed up the convergence of the iterative solver, preconditioning should be applied, which can be done by a simple Gauss-Seidel method or an incomplete LU decomposition for example. For further preconditioning techniques and mathematical derivation see [13].

Applying left-preconditioning to the system, the equation reads

\[ P^{-1} (AQ - b) = 0 \]  \hspace{1cm} (3.133)

Where \( P \) is called the preconditioning matrix. Compared to that, the right-preconditioning approach yields

\[ AP^{-1} \tilde{Q} = 0 \quad P^{-1} \tilde{Q} = Q \]  \hspace{1cm} (3.134)

Note that both possibilities are equivalent to the original system of equations \( AQ = b \).
ILU
The *incomplete LU* decomposition (ILU) is a common approach for preconditioning of any linear system of equations like \(3.118\). It computes an approximate factorization of the matrix \(A\) as follows

\[
A = LU + F \approx LU
\]

where \(L\) is a lower triangular matrix, \(U\) an upper triangular matrix and \(F\) an error term. In terms of the previous explanations, we then use \(P_{ILU} = LU\) as a preconditioning matrix. The approximation \(LU\) is easy to invert as the factorization leads to two simple triangular solves. In its simplest form, the ILU decomposition approximates the matrix \(A\) using the specific sparsity pattern of the matrix. Thus, only the entries \(l_{i,j}, u_{i,j}\) that correspond to an \(a_{i,j} \neq 0\) are computed. Due to that, the memory storage needed for the factorization is equal to that of the system matrix itself.

The implementation details can be found in [13] and [18]. There also exist more advanced versions of the same type, for example the ILUT where some additional values are calculated based on their influence inside the matrix \(A\) (see [16]).

SSOR
Another possibility is to use a Gauss-Seidel solver. Therefore we introduce the preconditioning matrix \(P\) as

\[
P_{\text{GS}}^{-1} = (D + L)^{-1}
\]

Here \(D\) is the diagonal part of the system matrix \(A\) in equation \(3.118\) and \(L\) is the lower part respectively.

The application of \(P_{\text{GS}}^{-1}\) to a vector \(x\) is easy to perform, as the computation of every entry \(x_{k}^{m+1}\) only requires values from the last iteration and already calculated values of the new iteration

\[
x_{k}^{m+1} = \frac{1}{a_{k,k}} \left( b_{k} - \sum_{i=1}^{k-1} a_{k,i} \cdot x_{i}^{m+1} - \sum_{i=k+1}^{n} a_{k,i} \cdot x_{i}^{m} \right)
\]

As this method of preconditioning is usually very slow, we can make use of the so called *successive over-relaxation* (SOR). Therefore we introduce a new parameter \(\omega\). The next value inside the Gauss-Seidel algorithm is then computed using a combination of the old value \(x_{k}^{m}\) and the new value \(x_{k}^{*}\) which was obtained using equation \(3.133\) as follows

\[
x_{k}^{m+1} = (1 - \omega) x_{k}^{m} + \omega x_{k}^{*}
\]

which is equivalent to the choice

\[
P_{\text{SOR}}^{-1} = \omega(D + \omega L)^{-1}
\]

Stability theory (for example von Neumann analysis [8]) tells us, that \(\omega\) has to be chosen between 0 and 2 to ensure convergence, where \(\omega \in (0, 1)\) is called *under-relaxation*.
and \( \omega \in (1, 2) \) is called over-relaxation respectively. For some highly non-linear equations under-relaxation might be necessary to obtain convergence at all, but we are interested in over-relaxation to accelerate the convergence of the Gauss-Seidel method. As will be shown later, the choice of \( \omega \) has a large influence on the computational time of the linear solver (see [5]).

Because of its simplicity, the Gauss-Seidel preconditioner may have to be applied more than one time to ensure convergence. It is possible to do many iterations of the Gauss-Seidel method in order to approach the solution further.

The symmetric SOR method makes use of the preconditioning matrix (see [13] for details)

\[
P_{SSOR}^{-1} = \omega(2 - \omega)(D + \omega U)^{-1}D(D + \omega L)^{-1}
\]

which means that one first performs a forward sweep using the SOR mentioned above and afterwards does a reverse sweep using the upper part \( U \) of the system matrix. This usually leads to a better approximation of the system matrix.

### 3.8 Compressed Row Storage

The system matrix is typically sparse, as it only includes values corresponding to the stencil neighbors. For a large number of unknowns, we do not want to store all the additional zeros of the matrix, because the memory storage increases quadratically with the number of total variables of the problem. To remedy this, we store the matrix in the so called compressed row storage format (CRS). This means that we define three vectors \( \text{data} \), \( \text{col_idx} \), \( \text{row_ptr} \). The data vector contains all the non zero entries of the matrix as they are traversed in a row-wise way and \( \text{col_idx} \) stores the corresponding column indices of each single entry. The entries in \( \text{row_ptr} \) vector indicate at which data value a new row starts. The last entry additionally stores the number of non zero elements in the matrix.

For example, we consider the following matrix

\[
\begin{bmatrix}
10 & 0 & 0 & 0 \\
3 & 9 & 0 & 0 \\
0 & 7 & 8 & 7 \\
3 & 0 & 7 & 8 \\
\end{bmatrix}
\]

The vectors for the CRS format would then look like

\[
\begin{align*}
\text{data} & = \{10, 3, 9, 7, 8, 7, 3, 7, 8\} \\
\text{col_idx} & = \{1, 1, 2, 2, 3, 4, 1, 3, 4\} \\
\text{row_ptr} & = \{1, 2, 4, 7\}
\end{align*}
\]

Depending on the programming language, the indices could also start with 0, shifting the \( \text{col_idx} \) and \( \text{row_ptr} \) values.
The amount of memory needed for storing a matrix $A \in \mathbb{R}^{n \times n}$ containing $m$ non zero elements can be easily calculated as $2 \cdot m + n$ compared to $n^2$ entries in the standard format. As we have a nine-point stencil, we end up with roughly $m = 9 \cdot n$ leading to $19 \cdot n$ memory storage. On a $40 \times 40$ grid with 1600 unknowns, this reduces the memory storage needed to almost 1%. It also speeds up the time needed for a standard matrix-vector product.
Chapter 4

Analysis

4.1 Stability Analysis

In general linear theory, implicit methods do not have to fulfill any stability constraints. Considering multi-block computations, the situation is slightly different, because the values inside each block might not be completely coupled with the other values. Thus, we have to make sure, that the decoupling does not lead to instabilities. For that purpose, we take a closer look at a simple two-block problem that we want to solve iteratively.

4.1.1 Model Problem

We consider the stationary version of the heat equation \([2.2]\). For simplicity we set

\[ \kappa(T) \equiv 1 \]  

resulting in a linear equation. We introduce a dual time derivative according to equation \([3.103]\). Furthermore we only use the one-dimensional version which leads to

\[ \frac{\partial T}{\partial \tau} - \frac{\partial^2 T}{\partial x^2} = 0 \]  

Now we discretize the equation using first order backwards differences in dual time and the standard second order discretization for the spatial derivatives (note that this approach is not the same as the discretization used by Kapper, but it makes it easier to get analytical results).

\[ \frac{T_i^{N+1} - T_i^N}{\Delta \tau} - \frac{T_i^{N} - 2T_i^N + T_i^{N+1}}{\Delta x^2} = 0 \]  

We define the coefficients in front of the different unknowns \(T_i^N\)

\[ a = -\frac{1}{\Delta x^2} \]  

\[ b = \frac{1}{\Delta \tau} + \frac{2}{\Delta x^2} \]  

\[ c = \frac{1}{\Delta \tau} \]
With \( \Delta x = \frac{1}{n+1} \) and \( n \) the number of interior grid points.

On the boundary we set Dirichlet boundary conditions \( T_0 \) and \( T_{n+1} \).

The classical single block approach therefore results in the following linear system of equations

\[
\begin{pmatrix}
  b & a \\
  a & b & \ddots \\
  \ddots & a & b & \ddots \\
  \vdots & \ddots & a & b & \ddots \\
  a & \ddots & \ddots & a & b \\
  \vdots & \ddots & \ddots & a & b \\
  a & \ddots & \ddots & a & b \\
  \vdots & \ddots & \ddots & a & b \\
  a & \ddots & \ddots & a & b \\
\end{pmatrix} \begin{pmatrix}
  T_1 \\
  T_2 \\
  \vdots \\
  T_{n-1} \\
  T_n \\
\end{pmatrix}^{N+1} = \begin{pmatrix}
  cT_1 - aT_0 \\
  cT_2 \\
  \vdots \\
  cT_{n-1} \\
  cT_n - aT_{n+1} \\
\end{pmatrix}^N
\]

(4.7)

For the multi-block calculations, we need to divide the numerical domain into parts. Here, we take two blocks and therefore divide the domain into halves. Every half is treated by one block and solved independently. After each iteration step, the boundary information is exchanged and the next iteration starts until the solution of both blocks has finally converged. During the simulation of a single block, the boundary values are thus fixed. We can still write down the system of equations for the whole domain by taking into account the special boundary treatment at the interior boundary between the two blocks as follows (for simplicity, we assume an even number of interior grid points \( n \))

\[
\begin{pmatrix}
  b & a \\
  a & b & \ddots \\
  \ddots & a & b & \ddots \\
  \vdots & \ddots & a & b & \ddots \\
  a & \ddots & \ddots & a & b \\
  \vdots & \ddots & \ddots & a & b \\
  a & \ddots & \ddots & a & b \\
  \vdots & \ddots & \ddots & a & b \\
  a & \ddots & \ddots & a & b \\
\end{pmatrix} \begin{pmatrix}
  T_1 \\
  \vdots \\
  T_n \\
\end{pmatrix}^N = \begin{pmatrix}
  cT_1 - aT_0 \\
  cT_2 \\
  \vdots \\
  cT_{n-1} \\
  cT_n - aT_{n+1} \\
\end{pmatrix}^n
\]

(4.8)

Note that the two additional zeros inside the matrix on the left hand side indicate the missing coupling of the blocks during one iteration.

As we are marching in dual time, we want to arrive at the next physical time level (which is the steady state in this case, as we are only considering the stationary heat equation), so we want to obtain the stationary solution values \( T_i^{\text{steady}} \) as

\[
\lim_{N \to \infty} T_i^N = T_i^{\text{steady}} \quad \forall i = 1, \ldots, n.
\]

(4.9)

Both methods from above can be summarized when using a parameter \( \omega \) for the coupling. \( \omega = 0 \) means that we use the classical one-block approach and \( \omega = 1 \) on the
other hand means, that we use the two-block approach without coupling. Intermediate values can be interpreted as a mixture of both methods.

This leads to the following system of equations

$$Ax^{N+1} = Bx^N + C$$  \hspace{1cm} (4.10)

With the definitions

$$x^N := \begin{pmatrix} T_1 \\ \vdots \\ T_n \end{pmatrix}^N$$  \hspace{1cm} (4.11)

$$A := \begin{pmatrix} b & a & & & \\ a & b & & & \\ & & a & & \\ & & b & a & (1 - \omega) \\ & & a(1 - \omega) & b & a & & & & \\ & & & & & & & a & (1 - \omega) & b & a \\ & & & & & & & b & a & & \\ & & & & & & & a & b & & \end{pmatrix}$$  \hspace{1cm} (4.12)

$$B := \begin{pmatrix} c \\ & & c \\ & & -a\omega \\ & & a\omega & c \\ & & & & & & & c \\ & & & & & & & & & & c \end{pmatrix}$$  \hspace{1cm} (4.13)

$$C := \begin{pmatrix} -aT_0 \\ 0 \\ \vdots \\ 0 \\ -aT_n \end{pmatrix}$$  \hspace{1cm} (4.14)

### 4.1.2 Analysis and Eigenvalue Computation

We want to analyze the equation system above with respect to its stability. Therefore we take a closer look at the stationary version of the equation (4.2) and set \(c = 0\) which is equal to an infinite dual time step \(\Delta \tau\) (from Section 3.5.2 we know that an infinite dual time step is equal to Newton’s method).
The new values $x^{N+1}$ are obtained by the update formulas

\begin{align}
A\tilde{x}^{N+1} = Bx^N + C \\
x^{n+1} = x^N + \alpha \cdot (\tilde{x}^{N+1} - x^N)
\end{align}

(4.15) (4.16)

This is the same as the Newton update, where the parameter $\alpha$ can be seen as a relaxation parameter that can accelerate the convergence in the instationary case. Typical values of $\alpha$ are between 0 and 2, where $\alpha < 1$ refers to a damping of the solution process due to stability problems and $\alpha > 1$ accelerates the convergence.

By inserting equation 4.10, we derive

\begin{align}
x^{N+1} &= x^N + \alpha \cdot (A^{-1} \cdot Bx^N + A^{-1}C - x^N) \\
&= [I + \alpha(A^{-1}B - I)] x^N + \alpha A^{-1}C \\
&= Mx^N + \alpha A^{-1}C
\end{align}

(4.17) (4.18) (4.19)

With iteration matrix $M = I + \alpha(A^{-1}B - I)$.

We now want to derive the stability limits with respect to the parameter $\alpha$ for different settings. In order to do so, we look at the propagation of errors. As every numerical calculation is error-prone, it is important to know whether these numerical errors are amplified or damped during the calculations. In the first case, the method is called instable, otherwise it is stable.

We define the error at step $N+1$ as

$$
\epsilon^{N+1} = x^{N+1} - x_h^{N+1}
$$

(4.20)

Where $x_h^{N+1}$ is the error-prone numerical solution and $x^{N+1}$ is the numerical solution obtained by exact calculations (e.g. without round-off errors).

As both solutions have been calculated using the same iteration 4.17 we obtain

$$
\epsilon^{N+1} = x^{N+1} - x_h^{N+1} = Mx^{N+1} - Mx_h^{N+1} = M\epsilon^N = M^N\epsilon^0
$$

(4.21)

We see, that we only need to look at the properties of the iteration matrix $M$ in order to determine stability regions for the error propagation as the matrix $M$ is applied to the initial error successively.

For the iteration matrix $M$ it is important to have a spectral radius $\rho$ smaller than one. The spectral radius is the largest absolute value of any eigenvalue of the matrix. Eigenvalues that have an absolute value larger than one lead to an amplification of the corresponding eigenvector component in $\epsilon$. Damping is achieved by eigenvalues with an absolute value smaller than one.

The spectrum of the iteration matrix $M$ can be obtained using a standard software like Mathematica. Because of the specific structure of the matrices $A$ and $B$, it is possible to get analytical results for arbitrary values of $\alpha$ and $\omega$. 
The eigenvalues of the iteration matrix $M$ are as follows

$$
\begin{align*}
\lambda_1 &= 1 - \alpha \frac{2n + 2}{2n + 2 - n\omega} \\
\lambda_2 &= 1 - \alpha \\
\vdots \\
\lambda_{n-1} &= 1 - \alpha \\
\lambda_n &= 1 - \alpha \frac{2}{2 + n\omega}
\end{align*}
$$


Where $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ for all $\omega$ and $n > 2$.

### 4.1.3 Stability Constraint

With the eigenvalues of the iteration matrix, it is possible to derive an interval for $\alpha$ such that the iteration is stable. As we want to damp error terms, the spectral radius of the iteration matrix $M$ has to be smaller than one. For the classical approach ($\omega = 0$) the fractions in (4.26) evaluate to one leading to a stable iteration for $\alpha \in (0, 2)$ which is the same result as for the overrelaxation parameter of the Gauss-Seidel solver, for example.

For the multi-block approach ($\omega = 1$), the largest eigenvalue is $\lambda_{\text{max}} = \lambda_1 = 1 - \alpha \frac{2n + 2}{n + 2}$. The stability constraint then leads to the condition

$$0 < \alpha < \frac{n + 2}{n + 1} = 1 + \Delta x$$

(4.27)

On the one hand, the result means that it is possible to run the simulation with $\alpha = 1$, but with increasing problem size due to finer grids, we approach the stability limit and the solution might heavily oscillate. For non-linear heat conductivity $\kappa = \kappa(T)$, the linear theory is not valid and the choice $\alpha = 1$ can in fact already lead to instable behavior.

An even more restrictive result can be obtained, if we demand that there should not be any oscillations and the solution should smoothly converge to the steady state. In that case, the eigenvalues have to be positive. Negative eigenvalues would change the direction of the corresponding error parts in every iteration step. We obtain positive eigenvalues if $\alpha$ satisfies the following condition

$$0 < \alpha < \frac{n + 2}{2n + 2} = \frac{1}{2} (1 + \Delta x)$$

(4.28)

As an example, we set $n = 10$ and want to compute the stability limits for Newton’s method as well as for the dual time stepping method. The maximum value of $\alpha$ ensuring stability for Newton’s method is $\alpha_{\text{max}} = \frac{n + 2}{n + 1} = \frac{12}{11} \approx 1.09091$, as we have seen above.

For the dual time stepping, $\alpha_{\text{max}}$ depends on the dual time step size $\Delta \tau$ because we add a value on the diagonal of the matrix $B$. We can do the same computation as in Section 4.1.3 again with $c \neq 0$, plot $\alpha_{\text{max}}$ with respect to $\Delta \tau$ and observe the resulting Figure 4.1.
We see, that we approach the value of the Newton method for $\Delta \tau \to \infty$, but for small values of $\Delta \tau$ we can allow a larger $\alpha$. We usually do not use an $\alpha$ for the dual time stepping, but the fact, that we could allow a large $\alpha$ reflects the better stability properties of the dual time stepping. In Section 3.5.2 we already mentioned, that this has to do with the additional value $\frac{1}{\Delta \tau}$ on the diagonal of the system matrix leading to improved diagonal dominance of the matrix.

Summarizing, we can say that it is possible to ensure the stability of the multi-block iterations by a careful choice of the parameter $\alpha$ in case of Newton’s algorithm. For the dual time stepping, the same is true for the dual time step size $\Delta \tau$ where small values significantly improve the stability.
Chapter 5

Results

The methods explained in Chapter 3 have been added to the existing framework. We now want to test the different parts of the extension to the framework to obtain good parameters and identify the most useful methods for the simulation of our test cases.

In the end, we test whether the new framework comes up with the runtime reduction we hope to get. To do so, we test the different methods bottom-up, meaning that we start with the linear and non-linear solvers and end with the time discretization.

We will only perform tests using the rather simple heat equation from Section 2.1 in order to concentrate on the numerical methods. Nevertheless, these tests reveal most of the interesting properties of the methods and are therefore suitable for our purpose.

5.1 Linear Solver

The large linear system of equations emerging from the discretization has to be solved numerically. In Section 3.7 we explained the techniques used to approach the solution iteratively. In order to get the different solvers running efficiently, we have to find the right parameters for some of the methods. We will now start discussing the effects of the different parameters on the quality of the solution and the associated runtime needed for the simulation. After that, we will be able to compare the solvers with respect to their runtime needed to solve the linear system.

As domain, we choose the unit square and we use three different grids with 40, 80 and 160 points in one dimension. On the boundary, we set homogeneous boundary conditions and choose initial condition \(T(x) = 1\). We solve the system emerging from the non-linear heat equation (2.2) with \(\kappa(T) = 0.5T^2\), the implicit Euler method with \(\Delta t = 0.01\) and \(\Delta \tau = 0.1\), meaning that we choose the dual time stepping as a non-linear solver. The results show values for the solution of the first linear system of equations. As all methods are iterative methods, we need a threshold for the residual which is chosen to be \(1 \cdot 10^{-8}\).
5.1.1 SSOR Preconditioner

Before working on the linear solvers themselves we have to consider the preconditioning because it is highly relevant for the runtime of the linear solvers as we will see later. So we start with examining the effect of the over-relaxation parameter $\omega$ on the SSOR-preconditioner. Figure 5.1 shows the results of a first test, using the SSOR method as a solver for the linear system of equations.

The residual is measured in the norm

$$\|u\| = \Delta x \sqrt{n \sum_{i=1}^{n} u_i^2}$$  \hspace{1cm} (5.1)

which is the discrete analogue to the $L^2$ norm

$$\|u\|_{L^2} = \sqrt{\int_{\Omega} u(x)^2 dx}$$  \hspace{1cm} (5.2)

The number of SSOR iterations for a residual threshold of $1 \cdot 10^{-8}$ is plotted against the over-relaxation parameter $\omega$.

![Figure 5.1: Iterations needed for the SSOR solver depending on the over-relaxation parameter $\omega$ for the three different grid sizes](image)

We can see, that the effect of $\omega$ on the number of needed iterations is quite impressive. The number of iterations decreases with increasing $\omega$ until the optimal value $\omega_{opt}$ is reached. The iterations are reduced by a factor of more than 100. After that, the residual is increasing again until the method is diverging for values of $\omega$ larger than 2. This behavior is typical for the SSOR method with over-relaxation, but the optimal value for $\omega$ depends on the problem. We can already see that $\omega_{opt}$ is larger for finer grids. In our test, we obtained the values from Table 5.1 for the optimal overrelaxation parameter and corresponding number of iterations.
We also see that the number of iterations is proportional to the number of unknowns in one dimension $n$. Let $m = n^2$ be the total number of interior points. As one iteration needs roughly $O(m)$ operations for our sparse system, we end up with a total effort of $O(n) \cdot O(m) = O(m^{3/2})$ which is a typical result for the SSOR method (In our test we also measure some memory overhead and time for the calculation of the residual leading to a larger solver time). For simple iteration methods without overrelaxation like the Jacobi method, one usually ends up with an effort of $O(m^2)$.

5.1.2 GMRES

After the investigation of the preconditioning we will now take a closer look at the GMRES solver as there is also a parameter involved, namely the number of iterations before a restart is done.

To ensure the right choice of the optimal overrelaxation parameter $\omega_{opt}$, we first repeat the previous test using the SSOR method (see 3.137) as preconditioner for the GMRES method without restarts for the same test settings.

![Figure 5.2: Iterations needed for the GMRES solver without restarts depending on the over-relaxation parameter $\omega$ for the three different grid sizes](image)

From the results in Figure 5.2 we see a similar behavior as before. The number of iterations decreases until $\omega_{opt}$ and increases again until we have divergence near $\omega \approx 2$. 

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\omega_{opt}$</th>
<th>iterations</th>
<th>time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.59</td>
<td>37</td>
<td>3.4</td>
</tr>
<tr>
<td>80</td>
<td>1.76</td>
<td>75</td>
<td>30</td>
</tr>
<tr>
<td>160</td>
<td>1.87</td>
<td>152</td>
<td>280</td>
</tr>
</tbody>
</table>

Table 5.1: Optimal overrelaxation parameter $\omega_{opt}$ and number of iterations for different grid size $n$ for SSOR solver
The optimal values have only slightly changed but the number of iterations shows a difference. We still achieve reductions of a factor larger than 100, but the number of iterations has increased compared to the first test and is no longer proportional to the grid size \( n \). Now the number of iterations is increased by a factor of 3 as the grid size increases by a factor of 2. This already indicates, that the SSOR method is possibly not the best preconditioner for our problem.

As the SSOR method itself is an iterative method, one could try to perform more than one iteration as a single preconditioning step and hope for a better behavior. This method did not lead to a substantial reduction in computational time in our case as the SSOR iterations are quite time consuming compared to a single GMRES iteration.

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \omega_{opt} )</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>1.59</td>
<td>297</td>
</tr>
<tr>
<td>80</td>
<td>1.77</td>
<td>900</td>
</tr>
<tr>
<td>160</td>
<td>1.88</td>
<td>2458</td>
</tr>
</tbody>
</table>

Table 5.2: Optimal overrelaxation parameter \( \omega_{opt} \) and number of iterations for different grid size \( n \) for GMRES solver

We continue with investigating the effect of the restarts on the number of iterations without preconditioning. We let \( m \) be the number of iterations before a restart is performed and change \( m \) to get an optimal value for this parameter. Therefore we have to distinguish between iteration count and time. The number of iterations will be increased for small values of \( m \) as the residual is only optimized in a low dimensional subspace, but the total solver time can still be smaller as the memory needed by the method is substantially smaller. For large values of \( m \) the allocation of memory becomes time consuming and has to be taken into account.

Figure 5.3 shows the result in two plots.

(a) iterations

(b) time

Figure 5.3: Effect of number of iterations before restart \( (m) \) on total number of iterations (left) and total solver time (right) for GMRES solver

The results show, what we already expected. The solution time first decreases due to
the reduced iteration count and in the end increases because of the memory allocation overhead. The optimal values for the different grids are presented in Table 5.3.

<table>
<thead>
<tr>
<th>n</th>
<th>m_{opt}</th>
<th>iterations</th>
<th>time (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>11</td>
<td>137</td>
<td>5</td>
</tr>
<tr>
<td>80</td>
<td>17</td>
<td>292</td>
<td>58</td>
</tr>
<tr>
<td>160</td>
<td>27</td>
<td>634</td>
<td>775</td>
</tr>
</tbody>
</table>

Table 5.3: Optimal values for \( m \), number of iterations and time for different grid size \( n \) for GMRES solver

The number of iterations now increases linearly with the problem size \( n \) and the time increases much faster. For \( N = n^2 \) the total number of interior points, the total time to solve the system can be approximated as \( O(N^{1.7}) \). For a very small number of iterations, one generally assumes the time to increase linearly with the number of interior points with a large proportionality constant. But as we see, the number of iterations depends on the number of interior points itself leading to a superlinear relation. Nevertheless, the iterative GMRES solver is still better than standard direct solvers which calculate the solution after \( O(N^3) \) operations. The solver time is still larger than for the simple SSOR method, which makes clear that the GMRES method is not suitable without a proper preconditioning method. As we will see in the next tests, the time is significantly decreased for the ILU preconditioner.

The time measurements are usually not so comparable, because we have to take into account memory allocation, cache effects, programming issues and compiler properties. Especially as we are using JAVA as programming language, the memory management has a large effect on the computational time because we have no possibility to take care of memory deallocation ourselves. Instead this is done by the garbage collector of the virtual machine. Additionally we also have some time overhead due to the object orientation of the programming language. The single solvers are encapsulated in classes from which the special methods have to be called. Therefore, a simple SSOR solver might be faster just because it is implemented in the same class as the rest of the simulation.

5.1.3 Linear Solver Comparison

After fine tuning the SSOR preconditioner and the GMRES solver, we now want to compare the different solvers and preconditioners explained in Section 3.7. Again, we take the same test setting as before and compute the solution of the linear system using each solver (GMRES, BiCGSTAB, BiCG) with every preconditioner (SSOR, ILU).

During this tests, it showed up that the SSOR method is in fact not suitable as a preconditioner for our problems. As we have already seen for the GMRES solver, it increases the number of iterations and for the BiCGSTAB and BiCG method we did not even get an accurate solution after a reasonable number of iterations. Therefore, we will only present the results for the not preconditioned linear solvers and for the preconditioner ILU. The results are shown in Figure 5.4.
Figure 5.4: Number of iterations and solution time for different linear solvers with and without preconditioning for $40 \times 40$ (up), $80 \times 80$ (middle) and $160 \times 160$ grid (bottom). GMRES method is executed with numbers of iterations before a restart according to $m_{opt}$ from Table 5.3.
First of all, we see that the preconditioned methods always outperform the unpre-
conditioned methods. This shows the suitability of the incomplete LU factorization as
a preconditioner. Without preconditioners, the BiCGSTAB method shows the best re-

sults with respect to the number of iterations and time. The number of iterations can be
reduced to one third and the time is reduced to approximately one half of the GMRES
value.

The preconditioned methods show different results. For the coarse grid, the BiCGSTAB
still performs good compared to the GMRES method but with increasing problem size
the GMRES methods gives better results. The big advantage of the BiCGSTAB method
is, that it has no parameters and can be used in a black-box fashion, whereas the GM-
RES method needs some optimization with respect to the restarts. We have seen that
the runtime indeed depends on the iterations before a restart so that the results of this
test are only valid for the optimized values, which we used here.

For the test case above, the preconditioning leads to a major reduction of iterations
and runtime, though the factorization itself for the ILU takes some time. We also did
some other tests, with smaller physical time step size $\Delta t$ in which the number of iterations
was obviously smaller because the starting value is already close to the solution. In those
tests, the preconditioning was not so efficient, because the number of iterations was so
small, that the use of the ILU preconditiner did not lead to a further acceleration. It is
also important to say, that the additional time for the preconditioner in the case of ILU
is increasing linearly with the number of interior points. This is due to the factorization
process in the beginning. The additional memory can also lead to problems for large
matrices.

![Figure 5.5: Residual development for different linear solvers with and without precon-
ditioning for 80 × 80 grid](image)

We also want to show the development of the residual during the solution process. In
Figure 5.5 the residual is plotted against the current iteration for the same test settings
than before but for a smaller residual threshold to obtain more iterations. Note that the
Results

monitored residual inside the GMRES algorithm is always in the preconditioned space, thus it is not possible to get comparable results for the preconditioned GMRES method. The effect of the preconditioner can be demonstrated using the BiCG and BiCGSTAB method. Here we clearly see the iteration reduction due to a faster convergence and the residual is also decreasing smoother than before. We observe that the residual decreases more slowly but monotonically for the GMRES method. This is a special property of the GMRES method as it directly aims at minimizing the residual where the BiCG methods minimize different functionals related to the residual.

In the end, we decided to use the ILU preconditioned BiCGSTAB solver for the further tests, as it is parameter free, comes up with a small memory consumption independent of the number of iterations and shows good results with respect to the solver time needed for one linear solve.

5.2 Non-linear Solver

After finding the best linear solver and corresponding preconditioner available, we can now take a look at the two different non-linear solvers, the dual time stepping and Newton’s algorithm.

As test case, we take the same problem as specified in Section 5.1 (non-linear heat equation 2.2 with \( \kappa(T) = 0.5T^2 \)) and use the BiCGSTAB solver together with the ILU preconditioner to solve the linear systems.

Here we want to examine the effect of the different parameter choices on the number of iterations needed to solve the non-linear system. The parameters are the size of the dual time step \( \Delta \tau \) for the dual time stepping and the relaxation parameter \( \alpha \) for Newton’s method.

We therefore vary the specific parameter and monitor the number of iterations and the behavior of the residual. The results are shown in Figure 5.6.

For the dual time stepping, we want to converge to a steady state of the underlying modified equation. Similar to other time dependent equations, we see a fast residual decrease in the beginning followed by a slower reduction near the steady state. The Figure indicates, that the convergence rate is increased for larger dual time steps \( \Delta \tau \). As we know, the method itself converges to Newton’s method for infinitely large dual time step sizes. A dual time step with infinitely large time step size therefore corresponds to a single Newton step. Furthermore, we have no oscillations of the residual during the simulation which shows the stabilizing effect of the dual time stepping. The drawback is, of course, that it is slower than Newton’s method as we see in comparison to the second figure.

Here we see the residual development of Newton’s method for different values of the parameter \( \alpha \). For this particular type of equation, the choice \( \alpha = 1 \) seems to be optimal. The number of iterations to reach the residual threshold of \( 1 \cdot 10^{-7} \) is larger for all other values, whereas values smaller than one perform slightly better than the corresponding values larger than one. For increasing \( \alpha \), the residual starts to oscillate, as can be seen for \( \alpha = 1.5 \). As the simple choice \( \alpha = 1 \) gives the best results, we use this for our further
Figure 5.6: Residual development for dual time stepping depending on dual time step size $\Delta \tau$ (up) and Newton’s algorithm depending on parameter $\alpha$ (down).

Simulations and keep $\alpha$ constant. In Section 3.5.1 we also explained the line search to accelerate convergence. In applications where the evaluation of the right hand side is costly, this approach can lead to increased runtime. As we have no stability problems during our simulation, we decided to do without a line search. For other equations this may be different.

We also did tests for other grid sizes with almost the same results. The number of iterations almost stays constant or increases by one or two. This is mostly due to the effect of a larger initial residual because of the influence of the boundary conditions. Besides that, the results are the same.

Summarizing we see, that the number of iterations for the best parameters is smaller in case of the Newton method, which needs only 6 iterations compared to 8 iterations for the large $\Delta \tau$ dual time stepping method. We therefore decide to use Newton’s method.
with $\alpha \equiv 1$ in the next test cases.

5.3 Computation of the Jacobian

In 3.6 we described different approaches to compute the jacobian of the system inside of the program. All these methods were implemented and thoroughly tested. The three distinct methods are the analytical computation, the finite difference (FD) and the efficient finite difference (efficient FD).

As we suppose the analytical computation to be accurate to machine precision, we take this method as a reference for the other methods. Besides the accuracy we will also test and compare the runtime of the specific methods.

Again, we take the same test case as in Section 5.1. As the results are comparable for every time discretization, we only present those for the implicit Euler method with Newton solver for the non-linear system.

5.3.1 Accuracy of Finite Differences

In this section we investigate the effect of the parameter choice for $\epsilon$ with respect to the accuracy of the computation of the Jacobian. Due to that, we compute the Jacobian using the FD method with step sizes $\epsilon$ varying from $10^{-13}$ to $10^{-1}$. The error is computed using the element-wise 2-norm between the difference of the analytical computation and the FD method:

$$\|A\| = \sqrt{\sum_{i,j=1}^{n} a_{i,j}^2}$$  \hspace{1cm} (5.3)

The relative error with respect to $\epsilon$ is presented in Figure 5.7.

The first observation is, that the error is almost the same for all three grids, because we measure the relative error. The absolute error is larger for finer grids, because the matrix has more non-zero entries.

Beginning with a high error for extremely small values of $\epsilon$ due to large numerical round-off errors, we observe a decrease of the error until $\epsilon = 10^{-8}$. In fact, the minimum is located around $4.0 \cdot 10^{-8}$. After that, the error is increasing again and gets very high for large values of $\epsilon$ because of the large discretization error of the FD method.

Of course, the error near the optimal $\epsilon$ should be small enough to obtain good results throughout the simulation. The minimum relative error of $2.0 \cdot 10^{-8}$ shows, that the FD method provides the necessary accuracy.

Furthermore, it is interesting to see if the optimal value $\epsilon_{opt}$ can be derived beforehand using any of the approaches mentioned in section 3.6.

As the machine accuracy was calculated to be

$$\epsilon ps = 2.2 \cdot 10^{-16}$$  \hspace{1cm} (5.4)

We can derive a first estimate for $\epsilon_{opt}$ with the help of equation 3.114 as follows

$$\epsilon = \sqrt{\epsilon ps} = \sqrt{2.2 \cdot 10^{-16}} \approx 1.5 \cdot 10^{-8}$$  \hspace{1cm} (5.5)
Figure 5.7: Relative error between FD and analytical Jacobian for varying $\epsilon$ and different grids

Which is actually quite near the real minimum of $4.0 \cdot 10^{-8}$.

The second approach mentioned in equation 3.115 yields similar result. Considering the discrete $L^2$-norm to be relevant for the error and using a flow vector of size $\|Q\| = 1.0$ as we did in the test above starting from initial condition 1, the suggested value for $\epsilon$ is

$$\epsilon = \sqrt{\text{eps}} (\|Q\| + 1) \approx 1.5 \cdot 10^{-8} \cdot 2 \approx 3 \cdot 10^{-8}$$

(5.6)

We see, that both methods give values close to the actual optimum.

In the non-linear test case above, we see the discretization error dominating the numerical error for large values of $\epsilon$. An additional test with the linear heat equation showed, that this discretization error in fact reduces to zero and we end up with an error of machine accuracy. For small $\epsilon$, the numerical round-off error still influences the accuracy to a great extend, but this additional error vanishes for larger values of $\epsilon$.

### 5.3.2 Runtime of Different Approaches

Besides the accuracy of the Jacobian calculation, we also want to compute the matrix as fast as possible to accelerate the whole simulation. Note that the Jacobian has to be computed in every single dual time step or Newton step respectively. To investigate the runtime of the different methods, we performed another test in which we measured the average time to calculate the Jacobian for the analytical method and the two finite difference methods.
Table 5.4: Runtime (in msec) for different Jacobian calculations on different grids

<table>
<thead>
<tr>
<th>Method</th>
<th>time $40 \times 40$</th>
<th>time $80 \times 80$</th>
<th>time $160 \times 160$</th>
</tr>
</thead>
<tbody>
<tr>
<td>analytical</td>
<td>3.30</td>
<td>13.06</td>
<td>56.68</td>
</tr>
<tr>
<td>efficient FD</td>
<td>6.27</td>
<td>39.78</td>
<td>158.09</td>
</tr>
<tr>
<td>FD</td>
<td>937.09</td>
<td>24350.24</td>
<td>416057.27</td>
</tr>
</tbody>
</table>

The results are shown in Table 5.4. We clearly see, that the runtime for the analytical method increases quadratically with the number of points in one dimension, i.e. linearly with the total number of interior points. The efficient finite differences approach needs more time but has roughly the same asymptotic, as the time increases by a factor of 4 from the $80 \times 80$ grid to the $160 \times 160$ grid. The standard finite differences method, which gives exactly the same numerical values for the Jacobian as the efficient version, needs much more time and the time is increasing faster. Comparing the last two columns, the time increases by a factor of 16, which means that the runtime is proportional to the number of points in one dimension $n$ to the power of 4. This is due to the fact, that this method needs $n^2$ calls of the right hand side function, which itself needs $\mathcal{O}(n^2)$ operations as it computes values for each interior variable. The efficient FD method on the other hand always needs 9 function evaluations because of the stencil size. The effort is therefore proportional to $9 \cdot n^2$.

As the efficient finite differences can be used in a black-box fashion for every equation and discretization as soon as the stencil of the method is known, we prefer this method. The analytical Jacobian might give exact results up to machine precision, but the accuracy of the finite differences has been shown to be sufficiently good. Therefore, we will continue to use the efficient finite differences method for the following tests.

5.4 Time Discretization

The next test cases involve the time discretization methods described in Section 3.3. It is important for us to verify the accuracy of the schemes and interrelate this with the complexity of the method.

5.4.1 Linear Test

In order to determine the accuracy of the methods, we did a test with the linear heat equation (2.6) including right hand side function $f(x) = 10 \cdot \sin(\pi x)$ as specified in Section 2.2. As we want to end up with a quasi one-dimensional problem, we set Neumann boundary conditions on two opposing boundaries and homogeneous boundary conditions on the remaining two boundaries. For the initial condition, we also take $T(x) = 0$.

This equation can be solved analytically for all times $t$ as explained in Section 2.2. Now, we calculate the numerical solution using one of the time discretization schemes and stop the simulation at $t_{\text{end}} = 0.1\text{sec}$. Then we calculate the discrete values of the exact
solution on the grid and calculate the discrete $L^2$ error. With vanishing physical time step size $\Delta t$, the time discretization error goes to zero and only the spatial discretization error is left.

![Graph showing relative error over time steps for different methods](image)

Figure 5.8: Relative error between numerical and analytical solution at time $t_{end} = 0.1$ for different time discretizations and number of physical time steps on $160 \times 160$ grid

Figure 5.8 shows the results of this first linear test. In general, we can classify the methods in three groups: The implicit Euler method is the only first order method. The trapezoidal method, the BDF2 methods (starting with Euler or trapezoidal method in the first step) and the Richardson extrapolation for the implicit Euler method are of second order in time and last but not least the fourth order Richardson extrapolation applied to the trapezoidal rule. The order of each single method is equivalent to the negative slope of the line in Figure 5.8 before the stationary value for small time steps is reached.

The error of the spatial discretization is approximately $1.5 \cdot 10^{-5}$. We furthermore see, that the higher order methods converge faster and therefore reach this error earlier than the lower order methods. The implicit Euler method for example has not even converged for the smallest time step sizes.

5.4.2 Non-linear Test

In the linear test case, each time step includes only one dual time or Newton iteration. Thus, we do a second test with the non-linear heat equation with $\kappa = 0.1 + 0.5 \cdot T^2$ and constant right hand side function $f(x) = 10$ starting from initial value $T(x) \equiv 0$. As an analytical solution of this equation for every time $t$ is not possible, we compute a very accurate numerical reference solution. In our case we choose the BDF2 method and a
very small time step size of $\Delta t = 1 \cdot 10^{-5}$. We additionally assume, that this solution is better than the other solutions of this test, so that the fine solution is really the reference. The advantage is, that this fine numerical solution now includes the spatial discretization error. By taking the difference to another coarser numerical solution, we can therefore measure the time discretization error directly.

When computing a reference solution on each grid, we obtained that the results for the relative error are independent from the grid chosen. This is due to the fact, that we only measure the error of the time discretization. Therefore, it is sufficient to test with the $40 \times 40$ grid. We first calculate the fine BDF2 solution until $t_{end} = 0.1$ and save the values. For every other method and time step size, we then also compute the solution at $t_{end} = 0.1$ and calculate the difference to the reference solution. The relative error is plotted against the number of time steps performed.

![Relative error between numerical and fine numerical solution at time $t_{end} = 0.1$ for different time discretizations and number of physical time steps on 40x40 grid](image)

Figure 5.9: Relative error between numerical and fine numerical solution at time $t_{end} = 0.1$ for different time discretizations and number of physical time steps on 40x40 grid

We observe the three different groups identified in the last test with the same convergence rates. The midpoint rule shows not the same asymptotic as the trapezoidal rule but comes up with a slightly larger error. The results are not spoiled by effects from the spatial discretization so that we see almost straight lines. Additionally, we see that the higher order methods really give smaller errors for small time step sizes. The Richardson extrapolation applied to the trapezoidal rule obviously is more accurate than the reference solution and the error therefore stays constant at around $10^{-9}$. But also the behavior for larger time step sizes is a bit different. We assume that this is due to the fact, that the error estimator is influenced by some small oscillations that are usual for the trapezoidal rule. As has been said, oscillations are damped very slowly.
Results

for the trapezoidal rule because it is not L-stable, thus we still see oscillations when we use very few time steps. This also spoils the accuracy of the error estimator because the oscillations interfere with the oscillations of the smaller time step trapezoidal rule for the error estimate. For larger time steps, we see that the error decreases much faster again.

Additional tests showed, that the error of the physical time stepping methods has not changed for smaller residual thresholds for the linear and non-linear solvers. That means, that the solvers are sufficiently accurate for our purposes.

Summarizing we obtain the promised convergence rates for our different methods and see, that the higher order methods give smaller errors as expected.

5.5 Adaptive Time Stepping

In Section 3.4, we explained various options to change the time step size according to an error estimate. We now want to test the different methods using a special test case from engineering application. For plasma flows, the equations contain right hand sides which largely depend on the flow variable itself as follows

\[
\frac{\partial T}{\partial t} + \lambda \frac{\partial^2 T}{\partial x^2} = f(T) \tag{5.7}
\]

At a certain time, the flow ignites due to a sudden change in the value of the right hand side. For example, the right hand side function could look like

\[
f(Q) = \text{min} + \frac{\text{max} - \text{min}}{1 - \text{erf}(\frac{Q - Q_0}{\text{eps}})} \left( \text{erf}(\frac{Q - Q_0}{\text{eps}}) - \text{erf}(\frac{0 - Q_0}{\text{eps}}) \right) \tag{5.8}
\]

Where \(Q_0\) is the ignition temperature, \(\text{min}\) is the minimum function value for temperature values smaller than \(Q_0\) and \(\text{max}\) is the maximum function value for values larger than \(Q_0\). \(\text{eps}\) is a smoothing parameter. For large values of \(\text{eps}\) the change in \(f\) is rather smooth, but for small \(\text{eps}\), the change approximately becomes a jump. The values we used in our test are \(Q_0 = 0.15, \text{max} = 20.0, \text{min} = 2.0, \text{eps} = 0.01\) leading to a sharp jump from 2 to 20 at \(Q_0 = 0.15\). In contrast, we choose constant heat conductivity \(\kappa = 1\). The test is done on the unit square domain using the \(40 \times 40\) grid.

Because of this right hand side function, there are several time scales involved now. The equation first seems to converge to the steady state, but once some values exceed the ignition temperature, the solution rapidly changes until the ignition has spread into the computational domain. After this short but fast changes, the solution really converges slowly to its steady state with respect to the higher right hand side function values. Due to these different time scales, the equation is very useful to test the adaptive time stepping schemes, as the time step size needs to adjust itself to the different time scales during calculations. So we expect rather large time step sizes before the ignition and in the end of the simulation on the one hand and very small time step sizes during the ignition process.
We first test the implicit Euler method and an integral controller with $k_I = 1.0$ and change the safety factor $\theta$ from 0.7 to 1.0.

![Graphs showing step size development with respect to simulation time for implicit Euler method and integral controller for different safety factors $\theta$. Red dots indicate rejected steps.](image)

Figure 5.10: Step size development with respect to simulation time for implicit Euler method and integral controller for different safety factors $\theta$. Red dots indicate rejected steps.

Comparing the different plots in Figure 5.10, the first observation is, that the number of rejected steps increases with the safety tolerance. Starting from 28 rejects for $\theta = 0.7$, 34 for $\theta = 0.8$ and 49 rejects for $\theta = 0.9$ the number dramatically increases up to 2331 for $\theta = 1.0$. The step rejects mainly occur at times close to the ignition. In this region, the error usually increases due to the change in the right hand side function. The adaptive time stepping methods have problems predicting a useful time step size because the assumption of slow variation is not fulfilled there. Thus the error becomes strongly non-linear leading to underestimation of the error and thus rejected steps. In regions with smooth right hand side function, we do not see any rejections.

In the beginning, the time step also needs to be changed, because one usually does not know the right time step with respect to the error bound. Therefore, a few steps are rejected at the very start of every simulation.

Besides that, the step size follows our intuition. It decreases during the ignition and increases before and after. We also obtain some oscillations for $\theta = 1.0$ close to the ignition. The total number of steps performed is smallest for $\theta = 0.9$ followed by the $\theta = 0.8$ case, because these parameter choices balance between small time steps and step rejection. For further tests, we chose $\theta = 0.8$ because it results in few step rejects.
together with not too small step sizes. This value is also used by SöDERLIND [20].

Figure 5.11: Step size development with respect to simulation time for implicit Euler method and different controllers. I-Controller up, PI-Controller middle. PC-Controller down. Red dots indicate rejected steps.

<table>
<thead>
<tr>
<th>controller</th>
<th>timesteps</th>
<th>rejected steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1170</td>
<td>34</td>
</tr>
<tr>
<td>PI</td>
<td>1169</td>
<td>32</td>
</tr>
<tr>
<td>PC</td>
<td>1128</td>
<td>14</td>
</tr>
</tbody>
</table>

Table 5.5: Number of timesteps and rejected steps for different controllers in test of Figure 5.11

We have used the same test settings to compare different controllers. Figure 5.11 shows the time step size with respect to simulation time for three different methods with certain parameters using a safety factor of $\theta = 0.8$. The number of rejected steps is smallest for the predictive controller PC with parameters $k_E = 0.4$ and $k_R = 0.7$. This method also uses the smallest amount of time steps in total to compute the solution at time $t_{end} = 0.4$ as can be seen in Table 5.5. Furthermore, we obtain a smoother behavior of the time step sizes compared to the other two methods. This is important, because it
reduces the oscillations and leads to less rejected steps. Besides the fact, that we need to have access to the error estimates of the last two steps, the PC controller calculates the new step size with only a very small additional time compared to the standard I or PI controller. For tests with other right hand side functions, e.g. the non-linear test in Section 5.4.2 we did not observe such a decrease in the number of time steps performed. For the last test, we will therefore use the simple I controller.

5.6 Comparison with Explicit Method

After having investigated the different parts of the project and identified the most useful solvers and parameters, for example, we now want to put everything together and compare the outcome with the results for an explicit solver.

Our aim is to solve the same equation in minimal time subject to a certain error bound, such that the solution is useful. For the equation, we take the non-linear heat equation with \( \kappa = 0.1 + 0.5 \cdot T^2 \) and constant right hand side function \( f(x) \equiv 10 \) like in the non-linear test case in Section 5.4.2. We start at time \( t_{\text{start}} = 0 \) from initial condition \( T \equiv 0 \) and want to calculate the solution at time \( t_{\text{end}} = 1 \). The test is carried out on the 40 \( \times \) 40 grid.

The explicit reference method is the explicit Euler method, explained in Section 3.3. As the explicit method has to fulfill a CFL constraint, it is not possible to choose a time step larger than a certain value. For the particular method and the standard spatial discretization resulting in the Laplacian matrix, the CFL number is denoted as

\[
CFL = \kappa \cdot \frac{\Delta t}{\Delta x^2} < \frac{1}{4} \tag{5.9}
\]

This definition of the CFL number results from a linear stability analysis and we only assume, that the corresponding result holds also in the non-linear case. Nevertheless, we will get a good approximation to the maximum step size as we will see.

As written above, the CFL number has to be smaller than one fourth. The maximum value of \( \kappa \) depends on the solution itself and can be estimated as \( \kappa = 0.1 + 0.5(1.4)^2 \) from tests with other methods. On the smallest grid, this leads to a maximum time step of \( \Delta t_{\text{max}} = 1.5625 \cdot 10^{-4} \). For our discretization we tested different \( \Delta t \) and obtained a maximum time step size of about \( \Delta t_{\text{max}} = 1.27 \cdot 10^{-4} \), which is close to the theoretical value. We therefore chose \( \Delta t = 1.0 \cdot 10^{-4} \) for the time step in our explicit method.

We also made error calculations for this step size in the setting of the non-linear test case from above (see Section 5.4.2) which showed that the error is slightly larger than the error of the implicit Euler method (with the same time step size) and has a value of about \( 3 \cdot 10^{-4} \). It is important to say again, that we cannot calculate with larger time steps even if we were satisfied with a larger error, because the stability of the explicit method is restricted by the CFL condition.

As the explicit Euler method is a first order method, it would not be fair to compare it to the higher order methods like BDF2. Therefore we test against the implicit Euler method and an implicit Euler method with adaptive time stepping. Additionally, we
show the resulting runtime for the Euler Richardson extrapolation to demonstrate the
good performance of higher order methods.

The implicit methods do not have to satisfy a stability constraint. The time step is
now only limited by the error bound. We aim at a relative error of about 1% and choose
the time step according to the non-linear test in which we measured the relative error
with respect to the time step size. For the implicit Euler method, the step size can be
chosen to be $\Delta t = 1 \cdot 10^{-2}$ and for the Richardson version, we can even allow $\Delta t = 1 \cdot 10^{-1}$.
For the adaptive time stepping, we choose an initial time step of $\Delta t = 1 \cdot 10^{-4}$ and let
the further time step sizes be chosen by an integral controller with parameters $k_I = 1.0$
and a safety factor of $\theta = 0.9$.

According to a relative error of about 1%, we should choose the tolerance to be of
that size, too. As we are measuring the error of the coarse method but using the finer
method, we can allow for larger tolerances and still get an error of about 1% in the end.
For the implicit Euler method, this allows for $\text{TOL} = 0.2$. Using the error estimate to
perform a Richardson extrapolation on top of the adaptive time stepping even lets us
compute with $\text{TOL} = 1.0$ as the accuracy of the method is much higher than for the
simple implicit Euler method.

For the implicit methods, we also test a frozen Jacobian version, where we only
compute the Jacobian at the beginning of the solution of the non-linear system and
keep the matrix constant as long as the residual is decreasing. Thus, we do not have
to compute a new Jacobian at every Newton iteration but we usually have to do more
iterations as the Jacobian refers to an older solution vector.

The resulting runtime for the different approaches is shown in Table 5.6 and Figure
5.12 shows the results for methods without frozen Jacobian approach. Table 5.6 also
shows the errors of the different methods as obtained from the test in Section 5.4.2.

<table>
<thead>
<tr>
<th>method</th>
<th>runtime (sec)</th>
<th>relative error</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit Euler</td>
<td>18.278</td>
<td>0.0313%</td>
</tr>
<tr>
<td>implicit Euler</td>
<td>2.513</td>
<td>1.132%</td>
</tr>
<tr>
<td>implicit Euler frozen</td>
<td>2.157</td>
<td>1.132%</td>
</tr>
<tr>
<td>ATS implicit Euler</td>
<td>2.012</td>
<td>1.095%</td>
</tr>
<tr>
<td>ATS implicit Euler frozen</td>
<td>1.653</td>
<td>1.095%</td>
</tr>
<tr>
<td>Richardson Euler</td>
<td>0.684</td>
<td>1.217%</td>
</tr>
<tr>
<td>Richardson Euler frozen</td>
<td>0.814</td>
<td>1.217%</td>
</tr>
<tr>
<td>ATS Richardson Euler</td>
<td>0.7427</td>
<td>1.068%</td>
</tr>
<tr>
<td>ATS Richardson Euler frozen</td>
<td>1.156</td>
<td>1.068%</td>
</tr>
</tbody>
</table>

Table 5.6: Runtime comparison for different implicit methods with the original explicit
approach and corresponding relative errors

Taking a first look at Table 5.6, we see that all methods indeed result in an error of
about one per cent. The only exception is the explicit Euler method that needs a much
smaller time step size for stability reasons. Therefore we also end up with a smaller
error.
We observe a much smaller runtime for all of our implicit methods. The explicit Euler method takes almost 20 seconds, whereas the fastest implicit methods need only less than one second. This is mainly due to the effect of a reduced number of physical time steps because of the missing stability constraint. Every single time step is much slower for the implicit methods, as the computation of the Jacobian and the non-linear solver as well as the linear solver require many operations. But the reduced number of steps outmatches this disadvantage and in the end leads to a better performance. Already the implicit Euler method reduces the runtime significantly, but the Richardson version gives even better results. The number of time steps is reduced for this method as it is a higher order method, but the error estimate requires two small steps in every single physical iteration.

Furthermore, we see that the adaptive time stepping gives similar results compared to the optimized implicit Euler and Richardson method.

On the one hand, the process of estimating the new time step size including different controller types, checking for step rejects and so on creates some overhead that slows down the adaptive method a bit.
On the other hand, it is important to know, that we only end up with comparably small runtimes for the non adaptive methods, as the step sizes are chosen with the help of results from previous tests. In particular, the time step sizes were chosen such that the error computed in Section 5.4.2 is approximately one percent. We therefore use a-priori information about the error that is usually not available in real applications. Instead, the adaptive time stepping could be used in a black-box fashion where we only have to chose the error threshold and we do not need to execute additional tests to calculate the time step size. In case of a much too large initial time step size, the first few steps will be rejected and the method then carries on with smaller step sizes leading to an accurate solution.

Another important benefit of the adaptive time stepping method is, that it also provides us with a very important property which is the stability of the method. Even in the non-linear case, we can therefore be sure to get reasonable results without prior knowledge about the behavior of the error. This is a great advantage compared with other explicit and implicit methods which may have stability problems in strongly non-linear regions.

As for the frozen Jacobian approach, we see that this only pays out for the adaptive time stepping method and the simple implicit Euler method, but it does not reduce runtime for the Richardson version. The runtime for the computation of the Jacobian has decreased, but the frozen Jacobian leads to much more non-linear iterations which in turn lead to a larger runtime. The relative error is still the same, as the non-linear solver has converged using the same residual threshold.

As a summary of this test, we can say that the implicit methods have shown that they perform better than the explicit method even though they require a lot of work concerning implementation. In the end, the runtime has been decreased by about 90% with the help of the Richardson method.

### 5.7 Spatial Discretization

As an addition, we can use the linear test from Section 5.4.1 to check the accuracy of the spatial discretization. Performing the test on the three different grids, we obtain the corresponding errors of the spatial discretization. The values are shown in Table 5.7

<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{error}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>$2.12 \cdot 10^{-4}$</td>
</tr>
<tr>
<td>80</td>
<td>$5.3 \cdot 10^{-5}$</td>
</tr>
<tr>
<td>160</td>
<td>$1.3 \cdot 10^{-5}$</td>
</tr>
</tbody>
</table>

Table 5.7: Relative error resulting from spatial discretization for different grid sizes

Obviously, the error decreases inversely proportional to $n^2$, leading to a second order spatial discretization. We also see, that the resolution of 40 points in one dimension is already sufficient to end up with an relative accuracy of less than 0.1% which underlines the quality of the spatial reconstruction.
Chapter 6

Conclusion

We have developed an implicit framework that can in general be applied to any PDE. First, we successfully tested the linear solver and investigated the runtime of different solver and preconditioner combinations. Here we identified that the BiCGSTAB method together with the incomplete LU decomposition shows a good performance.

Concerning the non-linear system, we compared the dual time stepping method and Newton’s method. We clearly saw, that the dual time stepping method converges to Newton’s method for large dual time steps. The analytical computation of the stability properties for both methods led to the conclusion, that the dual time stepping method is more stable in the context of a multi-block framework.

For the computation of the Jacobian, we implemented an analytical version as well as two finite differences versions. After the investigation of the optimal step size $\epsilon$, we showed that the efficient finite differences approach only leads to a small runtime overhead compared to the analytical method and performs much better than the standard finite differences approach. The use of our method allows for easy addition of new methods, as it does not require an analytical Jacobian but uses the finite differences in a black-box fashion.

In the investigation of the various time stepping methods for linear and non-linear test cases we obtained the corresponding convergence orders and verified the error estimator for the Richardson extrapolation.

The adaptive time stepping was tested for a highly non-linear equation with three time scales. We could show the effect of the different controllers and parameters involved. The time step size has been adjusted depending on changes in the solution resulting in smaller a error in those regions.

With respect to the original explicit solver, we could get a significant speedup and the possibility to use larger physical time step sizes as the implicit methods do not need to fulfill any CFL constraints.

Further work on this project could be the implementation of new dedicated physical models, which should be easy due to the modular software design. One essentially has
to specify the numerical fluxes and all the other methods of the framework can be used automatically. Another possibility is to work on the parallelization of the methods. The framework already has a possibility to generate multi-block grids and the results from the stability analysis can be used to deal with stability issues concerning the synchronization between the blocks.
Appendix A

Analytical Computation of the Jacobian

\[ A_{i,0} = + (n_{0,x} \cdot (\kappa_1 \cdot M_{0,1,2}) + n_{0,y} \cdot (\kappa_1 \cdot M_{0,2,2})) \]
\[ - (0) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,4}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,4})) \]  
\[ (A.1) \]

\[ A_{i,1} = + (n_{0,x} \cdot (\kappa_0 \cdot M_{0,1,3} + \partial \kappa_{2,0} \cdot a_{2,1}) + n_{0,y} \cdot (\kappa_0 \cdot M_{0,2,3} + \partial \kappa_{2,0} \cdot a_{2,2})) \]
\[ - (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,2}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,2})) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,0}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,0})) \]
\[ - (0) \]  
\[ (A.2) \]

\[ A_{i,2} = + (0) \]
\[ - (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,3}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,3})) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,2}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,2})) \]
\[ - (0) \]  
\[ (A.3) \]

\[ A_{i,3} = + (n_{0,x} \cdot (\kappa_0 \cdot M_{0,1,0} + \partial \kappa_{0,0} \cdot a_{0,1}) + n_{0,y} \cdot (\kappa_0 \cdot M_{0,2,0} + \partial \kappa_{0,0} \cdot a_{0,2})) \]
\[ - (0) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,5}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,5})) \]
\[ - (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,4}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,4})) \]  
\[ (A.4) \]

\[ A_{i,4} = \frac{1}{\Delta t} + \frac{1}{\Delta t} + (n_{0,x} \cdot (\kappa_0 \cdot M_{0,1,1} + \partial \kappa_{0,1} \cdot a_{0,1}) + n_{0,y} \cdot (\kappa_0 \cdot M_{0,2,1} + \partial \kappa_{0,1} \cdot a_{0,2})) \]
\[ - (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,0} + \partial \kappa_{1,0} \cdot a_{1,1}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,0} + \partial \kappa_{1,0} \cdot a_{1,2})) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,1} + \partial \kappa_{2,1} \cdot a_{2,1}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,1} + \partial \kappa_{2,1} \cdot a_{2,2})) \]
\[ - (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,0} + \partial \kappa_{3,0} \cdot a_{3,1}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,0} + \partial \kappa_{3,0} \cdot a_{3,2})) \]  
\[ (A.5) \]
\[ A_{i,5} = + (0) \]
\[ - (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,1} + \partial \kappa_{1,1} \cdot a_{1,1}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,1} + \partial \kappa_{1,1} \cdot a_{1,2})) \]
\[ + (n_{2,x} \cdot (\kappa_2 \cdot M_{2,1,3}) + n_{2,y} \cdot (\kappa_2 \cdot M_{2,2,3})) \]
\[ - (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,2}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,2})) \]  
(A.6)

\[ A_{i,6} = + (n_{0,x} \cdot (\kappa_0 \cdot M_{0,1,4}) + n_{0,y} \cdot (\kappa_0 \cdot M_{0,2,4})) \]
\[ - (0) \]
\[ + (0) \]
\[ - (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,5}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,5})) \]  
(A.7)

\[ A_{i,7} = + (n_{0,x} \cdot (\kappa_0 \cdot M_{0,1,5}) + n_{0,y} \cdot (\kappa_0 \cdot M_{0,2,5})) \]
\[ - (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,4}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,4})) \]
\[ + (0) \]
\[ - (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,1} + \partial \kappa_{3,1} \cdot a_{3,1}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,1} + \partial \kappa_{3,1} \cdot a_{3,2})) \]  
(A.8)

\[ A_{i,8} = - (0) \]
\[ + (n_{1,x} \cdot (\kappa_1 \cdot M_{1,1,5}) + n_{1,y} \cdot (\kappa_1 \cdot M_{1,2,5})) \]
\[ - (0) \]
\[ + (n_{3,x} \cdot (\kappa_3 \cdot M_{3,1,3}) + n_{3,y} \cdot (\kappa_3 \cdot M_{3,2,3})) \]  
(A.9)

Here, \( n_{i,x} \) denotes the face normal of the \( i \)-th face of the center cell in x direction, \( M_{i,j,k} \) is the entry of the least squares matrix for the cell \( i \) in the \( j \)-th row and \( k \)-th column and \( A_{i,j} \) is the coefficient for the \( i \)-th equation of the system in front of the value that has the index number \( j \) inside the stencil of this cell. Furthermore \( \kappa_i \) is the heat conductivity of the face \( i \) averaged using formula (3.4) and \( \partial \kappa_{i,j} \) is the derivative of \( \kappa \) of the \( i \)-th face with respect to the cell with the index \( j \) inside the flux calculation stencil.

Notice there is a zero, if the cell does not contribute to the flux over a particular cell face. Furthermore, the physical time discretization adds the second term to the coefficient of the center cell \( A_{i,4} \).

If one of the fluxes for a certain cell is a flux over a boundary face, the calculations are a bit more difficult. This is due to the face that the solution from the least squares is different. In this case, one can just substitute the matrix \( M \) by another expression as follows
\[ \tilde{M} = M - N \cdot C \cdot M \]  
(A.10)

In this case, also the entries \( a_{i,j} \) have to be replaced by the boundary values \( \tilde{a}_{i,j} \).
References


