CHAIR FOR COMPUTATIONAL ANALYSIS OF TECHNICAL SYSTEMS

RWTH AACHEN UNIVERSITY

This work was submitted to the
Chair for Computational Analysis of Technical Systems

Diese Arbeit wurde vorgelegt am
Lehrstuhl für Computergestützte Analyse Technischer Systeme

Modernes Design eines C++17 Finite-Elemente Kontinuumsmechanik Simulations-Codes: Automatisierte Ableitungsbestimmung und flexible Kopplungsstrategien

Modern design of a C++17 finite element continuum mechanics simulation code: Automatized differentiation and flexible coupling strategies

Masterarbeit

Master-Thesis

von / presented by

Wolff, Daniel

344490

Erstprüfer / First Examiner	Univ.-Prof. Ph. D. Marek Behr
Zweitprüfer / Second Examiner	M. Sc. Florian Zwicke
communicated by	Univ.-Prof. Ph. D. Marek Behr

Aachen, 26.02.2020

Contents

Glossary

Greek Symbols

Γ	Boundary of the computational domain
η	Dynamic viscosity
κ	Thermal conductivity
ν	Index of a Newton iteration
ρ	Density
$\boldsymbol{\sigma}$	Cauchy stress tensor
ϕ	Ansatz function for FEM discretization (e.g. hat functions)
φ	Elemental function
Ω	Computational domain
Ω^e	A grid cell (element)
Ω^{h}	Computational grid
Ω_{ref}	The reference element

Latin Symbols

$\mathbf{1}$	Vector where every entry is one
$\mathcal{A}$	An arbitrary differential operator
c_p	Heat capacity at constant pressure
$\boldsymbol{e}_i$	i-th Cartesian unit vector
h	Step width
H^1	Sobolov space of once weakly differentiable functions which are also square-integrable
$\mathbf{Id}$	Identity matrix
$\mathbf{J}_{\boldsymbol{F}}$	Jacobian of a function $\boldsymbol{F}$
$\mathcal{L}^2$	Space of square-integrable functions
n	Index of a time level
n_{el}	Number of elements
n_{en}	Number of element nodes
n_{n}	Number of nodes
n_{sd}	Number of spatial dimensions
n_{ts}	Number of time steps
$\boldsymbol{n}$	Outer normal vector
$\mathcal{P}^q$	Space of q-th order polynomials
p	Pressure
q	Surface-related heat flow
$\boldsymbol{R}$	Residual vector containing the discretized nonlinear FEM formulation
Re	Reynolds number

$\mathcal{S}$	Solution space
T	Temperature
t	Time
$\boldsymbol{u}$	Vector containing the unknown nodal values of a FEM discretization
$\mathcal{V}$	Test space
w	Analytical test function
$\boldsymbol{x}$	Spatial coordinates

Indices and Superscripts

0	Quantity prescribed as initial condition
a	Quantity related to reverse (adjoint) mode of automatic differentiation
C	Quantity related to the continuity equation
D	Quantity prescribed on a Dirichlet boundary condition
e	Element-level quantity
end	Quantity corresponding to the last time step
H	Quantity related to the heat conduction equation
h	Discretized quantity
M	Quantity related to the momentum equation
N	Quantity prescribed on a Neumann boundary condition
p	Perturbed quantity (related to finite differences)
p	Quantity related to the pressure field
ref	Reference quantity
t	Quantity related to forward (tangent) mode of automatic differentiation
T	Quantity related to the temperature field
v	Quantity related to the velocity field

Acronyms

AD	Algorithmic Differentiation
CAMPIGA	Computational Analysis using Meta-Programming and Isogeometric Analysis, a continuum mechanics simulation code written in C++17
CD	Computational Differentiation
DAG	Directed Acyclic Graph
FD	Finite Differences

FEM	Finite Element Method
IGA	Isogeometric Analysis
NSE	Navier-Stokes Equations
ODE	Ordinary Differential Equation
PDE	Partial Differential Equation
SAC	Single Assignment Code
XNS	An FEM solver, written in FORTRAN

List of Figures

List of Tables

1. Introduction

Simulation is nowadays an important part of science and engineering. There is a multitude of software available, ranging from proprietary via open-source to purely internally used in-house codes. In many cases, the latter have been developed on a by-need basis over a long time. The usual consequence is a code that on the one hand is highly specialized for a specific application and on the other hand contains a variety of different programming paradigms. Therefore, the implementation of new features may turn out to be quite tedious and prone to errors; as the original software architecture was designed for efficiency and not for extensibility. Especially older programming languages such as FORTRAN or C, which were over years the state-of-the-art languages for scientific software, lack possibilities to design a modular and extensible architecture while at the same time retaining competitive computational performance.

To this end, the CATS institute of RWTH Aachen and the ILSB institute of TU Vienna develop the novel C++17 continuum mechanics simulation code CAMPIGA with the intention of solving structural and fluid mechanic problems using Isogeometric Analysis (IGA) or the Finite Element Method (FEM). A variety of modern programming techniques and data structures are applied, which allow writing code with a readable and clearly defined interface for users, while simultaneously ensuring computational efficiency through the use of compiler optimization techniques.

As mentioned before, there is also a lot of open-source software available, which is capable of solving partial differential equations (PDEs) using an FEM approach. A first – even though incomplete – overview is supposed to be provided by the *FEA-compare* project, which is publicly available on GitHub[1]. However, none of these simulation codes documented there comprises all the features that should be part of the new software tool, such as space-time finite elements, IGA, or an automatized determination of the Jacobian (e.g., using algorithmic differentiation (AD) or finite differences (FD)) which is needed to solve non-linear problems by Newton's method. Therefore, the development of a novel software seemed to be more flexible with regard to the research topics at the institute. C++ as programming language has been chosen due to its efficiency: It allows developing programs, whose performance is very close to pure machine code [17], while simultaneously offering an environment that assists developers with a larger set of programming techniques, data structures, and libraries.

The main part of this thesis focuses on the implementation of automatized differentiation into the novel CAMPIGA software, paying special attention to the efficiency compared to alternative approaches like a manual implementation of the derivative or a computation using finite differences. Less emphasis will be placed on how flexible coupling strategies can be realized within the software architecture of CAMPIGA.

[1] see https://github.com/kostyfisik/FEA-compare, state 02.2020

The thesis is structured in two parts. The first part concentrates on computational differentiation (CD). Chapter 2 will give an overview of the current state of the art techniques for obtaining derivatives in modern FEM simulation codes. The key ideas of manual, symbolic and numerical differentiation will be roughly explained before finally basic concepts of algorithmic differentiation will be introduced using illustrative examples. The following Chapter 3 will then deal with the efficient implementation of a framework for obtaining derivatives automatically using either AD or FD approaches. Another section will be dedicated to its integration into the already existing CAMPIGA software. Chapter 4 shows some test cases which have been simulated using the extended CAMPIGA library. A Poiseuille flow will be simulated and compared to the analytical solution, in order to verify that all implemented methods for assembling the Jacobian work correctly. After that, the performance will be analyzed and compared for the different assembly strategies and different mesh sizes. As a second test case a cavity flow will be examined. The obtained results are compared to results from the literature. The influence of the assembly type on the number of Newton iterations needed to obtain a converged result will be considered as well. The last test case will solve a sample problem using both the monolithic as well as the staggered approach. The two solutions can then be compared with respect to the accuracy of the obtained solution as well as the number of Newton iterations needed to converge. This completes the first part of this thesis.

The second part of this thesis will then deal with the solution strategies for coupled problems. Chapter 5 gives an introduction to coupled problems and presents different strategies for solving coupled PDE systems. The main focus will be a comparison of the monolithic approach, which aims to solve the whole system at once, and staggered approaches that rely on an iterative solution involving information exchange between smaller subproblems. For the sake of completeness, the basics of field elimination and fractional step methods will be shortly presented. The following Chapter 6 discusses the mathematical difference of the monolithic and the partitioned approach and shows, how a coupling framework has been integrated into CAMPIGA. In Chapter 7, the implementation is validated by a comparison with the in-house code XNS. An additional test case is simulated to compare the coupling approaches with respect to their convergence.

Finally, Chapter 8 will summarize the work done in this thesis before it ends with an outlook, which motivates the next steps for extending the presented research topics.

Part I.
Computational Differentiation

2. Overview of Computational Differentiation

The need for calculation of derivatives arises in many fields of scientific computing:

- In *sensitivity analysis* the change of a model's output $\boldsymbol{y} \in \mathbb{R}^n$ with respect to some input parameters $\boldsymbol{u} \in \mathbb{R}^m$ is examined. The model can be considered a mapping $\boldsymbol{f} : \mathbb{R}^m \to \mathbb{R}^n, \boldsymbol{u} \mapsto \boldsymbol{y} = \boldsymbol{f}(\boldsymbol{u})$ between the input space and the output space. In order to quantify these changes, one essentially needs to compute the gradient of the mathematical model with respect to its inputs, namely

$$\mathbf{S} = \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} = \frac{\partial \boldsymbol{f}(\boldsymbol{u})}{\partial \boldsymbol{u}} \,. \tag{1}$$

  Mathematically speaking, $\mathbf{S}$ corresponds to the Jacobian $\mathbf{J}_{\boldsymbol{f}}$ of the model with respect to its input parameters, which is defined as

$$\mathbf{S} = \begin{pmatrix} \frac{\partial y_1}{\partial u_1} & \cdots & \frac{\partial y_1}{\partial u_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial u_1} & \cdots & \frac{\partial y_n}{\partial u_m} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1(\boldsymbol{u})}{\partial u_1} & \cdots & \frac{\partial f_1(\boldsymbol{u})}{\partial u_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n(\boldsymbol{u})}{\partial u_1} & \cdots & \frac{\partial f_n(\boldsymbol{u})}{\partial u_m} \end{pmatrix} = \mathbf{J}_{\boldsymbol{f}} \,. \tag{2}$$

  Every row of the resulting matrix $\mathbf{S} \in \mathbb{R}^{n \times m}$ corresponds to the sensitivity of one model component with respect to all its inputs. Sensitivity analysis is applied in many different fields of research, including financial modeling [11, 8], weather modeling [6] and of course engineering [40].

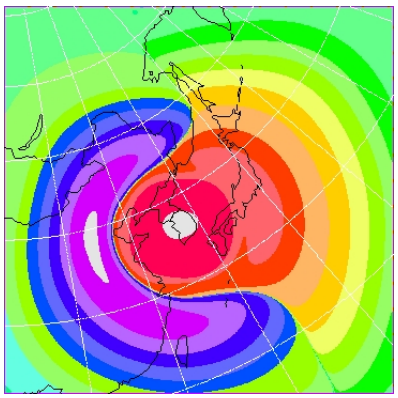


Figure 1: Sensitivities of the MM5 weather model with respect to the temperature. Image taken from [6].

- *Optimization* is another field of research, where the knowledge of derivatives is often very useful in order to solve related problems: In case of smooth objective

functions, gradient-based algorithms often have a superior efficiency – in terms of convergence and accuracy – compared to their derivative-free counterparts [24]. A simple example for a gradient-based algorithm is the steepest descent method [31], which aims to optimize an objective function $f : \mathbb{R}^n \to \mathbb{R}, \boldsymbol{x} \mapsto y = f(\boldsymbol{x})$ starting at some initial point $\boldsymbol{x}^0 \in \mathbb{R}^n$ by searching in the direction of the negative gradient in each iteration $k \in \mathbb{N}$:

$$\boldsymbol{x}^{k+1} = \boldsymbol{x}^k - \alpha^k \, \boldsymbol{\nabla} f(\boldsymbol{x}^k) \, . \tag{3}$$

Here, $\alpha^k \in \mathbb{R}^+$ represents an appropriately chosen step width, which may be used to scale the negative gradient in order to obtain better convergence. The gradient of f is defined as

$$\boldsymbol{\nabla} f(\boldsymbol{x}^k) = \left. \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right|_{\boldsymbol{x}^k} = \begin{pmatrix} \frac{\partial f(\boldsymbol{x}^k)}{\partial x_1} \\ \vdots \\ \frac{\partial f(\boldsymbol{x}^k)}{\partial x_n} \end{pmatrix} \, . \tag{4}$$

Fig. 2 shows the application of a simple steepest descent method to a two dimensional parabola.

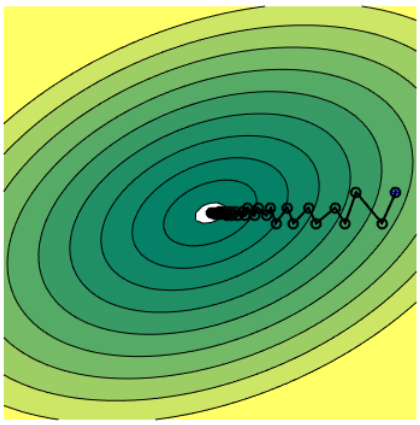


Figure 2: Iterative numerical solution of an unconstrained quadratic optimization problem using a steepest descent method. Darker colors indicate lower function values, the iteration points are marked with circles.

- The last example is the most relevant for this thesis: Derivatives are needed for the *iterative solution of nonlinear equation systems* using Newton's method [12]. A nonlinear equation system can be written in the residual form

$$\boldsymbol{R}(\boldsymbol{u}) = \boldsymbol{0} \, , \tag{5}$$

where the residual is a mapping $\boldsymbol{R} : \mathbb{R}^N \to \mathbb{R}^N, \boldsymbol{u} \mapsto \boldsymbol{R}(\boldsymbol{u})$. Applying a Taylor Series expansion about two successive iterands $\boldsymbol{u}^\nu$ and $\boldsymbol{u}^{\nu+1}$, one obtains

$$\boldsymbol{R}(\boldsymbol{u}^{\nu+1}) = \boldsymbol{R}(\boldsymbol{u}^\nu) + \mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu) \cdot \Delta \boldsymbol{u}^\nu + \mathcal{O}\left((\Delta \boldsymbol{u}^\nu)^T \Delta \boldsymbol{u}^\nu \right) \, , \tag{6}$$

where the Jacobian $\mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu)$ is defined as

$$\mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu) = \frac{\partial \boldsymbol{R}(\boldsymbol{u})}{\partial \boldsymbol{u}}\bigg|_{\boldsymbol{u}^\nu} = \begin{pmatrix} \frac{\partial R_1(\boldsymbol{u}^\nu)}{\partial u_1} & \cdots & \frac{\partial R_1(\boldsymbol{u}^\nu)}{\partial u_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial R_n(\boldsymbol{u}^\nu)}{\partial u_1} & \cdots & \frac{\partial R_n(\boldsymbol{u}^\nu)}{\partial u_m} \end{pmatrix} \in \mathbb{R}^{n \times m} \tag{7}$$

and $\Delta \boldsymbol{u}^\nu = \boldsymbol{u}^{\nu+1} - \boldsymbol{u}^\nu$. Evaluating Equation (5) at the newest iterand $\boldsymbol{u}^{\nu+1}$, plugging in Equation (6), truncating after the linear term and applying some algebraic transformations yields Newton's method. It reads:

$$\mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu) \cdot \Delta \boldsymbol{u}^\nu = -\boldsymbol{R}(\boldsymbol{u}^\nu)\,. \tag{8}$$

This is a linear equation system for the increment $\Delta \boldsymbol{u}^\nu$. From that the iterand of the next iteration can be computed using the update formula

$$\boldsymbol{u}^{\nu+1} = \boldsymbol{u}^\nu + \Delta \boldsymbol{u}^\nu\,. \tag{9}$$

Nonlinear equation systems appear in many scientific problem settings, e.g., when solving the FEM discretization of a nonlinear PDE system This will be discussed in more detail in Chapter 3.

After these introductory examples, the next section presents some methods for calculating derivatives in scientific code. According to [5], there are mainly four different ways to compute derivatives in scientific software, namely manual differentiation, symbolic differentiation, numerical differentiation, and algorithmic differentiation. Each of them will be presented in one of the following sections.

2.1. Manual Differentiation

One option is to compute the desired derivatives *manually*. This means to derive a symbolical expression for the derivative using the rules of differential calculus. The obtained results are then hard-coded into the developed software. If done correctly, this procedure is the fastest one [26]. The resulting terms can be simplified as much as possible and the implementation can be optimized in terms of the amount of floating-point operations needed to evaluate the derivative expression. Additionally, analytical derivatives are by construction exact (with respect to machine precision). However, this method is very prone to errors and time-consuming for non-trivial derivatives. Errors in the produced derivative code are hard to detect and the debugging process may take a long time. Additionally, this method does not allow to compute derivatives of functions, where a closed-form analytical expression is not available. Nevertheless this method forms a good starting point to approach the wide topic of computational differentiation (CD).

To obtain a strategy, how derivatives can be manually generated for scientific codes, consider the representation of mathematical functions in a programming language

such as C++. In C++, arithmetic operations are defined in terms of operators, which can be considered special functions. Each mathematical expression is divided into a series of operator or function calls, which are then performed one after another.

Listing 1: Complex number example using operator syntax

```
complex<double> foo(complex<double> x1, complex<double> x2) {
    return x1 - 2.0 * x2;
}
```

Listing 1 shows a small C++ code snippet, which computes the expression $x_1 - 2 \cdot x_2$ for arbitrary complex numbers $x_1, x_2 \in \mathbb{C}$. The compiler will replace the operator syntax by nested function calls similar to the code presented in Listing 2.

Listing 2: Complex number example using function syntax

```
complex<double> foo(complex<double> x1, complex<double> x2) {
    return operator-(x1,operator*(2.0,x2));
}
```

Motivated by this observation, we can rewrite the example introducing temporary variables such that each temporary variable stores the result of a single arithmetical or mathematical expression. The resulting so-called single assignment code (SAC) is shown in Listing 3.

Listing 3: Single assignment code for the complex number example

```
complex<double> foo(complex<double> x1, complex<double> x2) {
    complex<double> v1 = x1;
    complex<double> v2 = x2;
    complex<double> v3 = 2.0 * v2;
    complex<double> v4 = v1 - v3;
    return v4;
}
```

The fact that each (numerical) program can be decomposed into a SAC is one of the key strategies for generating derivatives. Let $\boldsymbol{F} : \mathbb{R}^m \to \mathbb{R}^n, \boldsymbol{x} \mapsto \boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$ be a differentiable function whose derivative should be computed. We assume that the implementation of $\boldsymbol{F}$ can be decomposed into a SAC such that for the intermediate variables v_j holds

$$v_j = \varphi_j(\{v_i\}_{i \prec j}), \tag{10}$$

with $j = m+1, \ldots, m+p+n$. φ_j represents an elemental function. An elemental function can be either an arithmetic operator (like $+, -, \cdot, \div, \ldots$) or a mathematical function

(like $\sqrt{(\cdot)}, \exp(\cdot), \ldots$). The notation $i \prec j$ means that the computation of v_j depends on the (intermediate) variable v_i, which also implies that $i < j$ holds. The range of the index j stems from the fact, that – similar to Listing 3 – all variables (not only the temporaries but also the inputs and outputs) are renamed and ordered in a large vector $\boldsymbol{v} \in \mathbb{R}^{m+p+n}$ with

$$\boldsymbol{v} = (\underbrace{v_1, \ldots, v_m}_{\boldsymbol{x}}, \underbrace{v_{m+1}, \ldots, v_{m+p}}_{\text{temporary variables}}, \underbrace{v_{m+p+1}, \ldots, v_{m+p+n}}_{\boldsymbol{y}})^T . \tag{11}$$

To understand this better, let us consider an example. The Rosenbrock function $f_{\mathsf{Rb}}(\boldsymbol{x}) : \mathbb{R}^2 \to \mathbb{R}, \boldsymbol{x} \mapsto f_{\mathsf{Rb}}(\boldsymbol{x})$ was presented in [34] and reads

$$f_{\mathsf{Rb}}(\boldsymbol{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 . \tag{12}$$

The gradient of the Rosenbrock function is given by

$$\boldsymbol{\nabla} f_{\mathsf{Rb}}(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial f_{\mathsf{Rb}}}{\partial x_1} \\ \frac{\partial f_{\mathsf{Rb}}}{\partial x_2} \end{pmatrix} = \begin{pmatrix} -400x_1(x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{pmatrix} . \tag{13}$$

One possible SAC for this function might look as follows:

$$\begin{aligned} v_1 &= x_1 \\ v_2 &= x_2 \\ v_3 &= v_1^2 \\ v_4 &= 1 - v_1 \\ v_5 &= v_2 - v_3 \\ v_6 &= v_4^2 \\ v_7 &= v_5^2 \\ v_8 &= 100 \cdot v_7 \\ v_9 &= v_8 + v_6 \\ y &= v_9 . \end{aligned} \tag{14}$$

A SAC can be visualized as a directed acyclic graph (DAG). The DAG for the Rosenbrock function is shown in Fig. 3. The nodes of the DAG correspond to the variables introduced by the SAC, while the edges visualize the dependencies between them according to Equation (10).

This representation of numerical functions forms one basis of computational differentiation strategies. The other key idea is the successive application of the chain rule to all (elemental) operations involved in the computation. For two differentiable functions $\boldsymbol{G}_{\mathsf{outer}} : \mathbb{R}^l \to \mathbb{R}^n, \boldsymbol{z} \mapsto \boldsymbol{G}_{\mathsf{outer}}(\boldsymbol{z})$ and $\boldsymbol{G}_{\mathsf{inner}} : \mathbb{R}^m \to \mathbb{R}^l, \boldsymbol{x} \mapsto \boldsymbol{G}_{\mathsf{inner}}(\boldsymbol{x})$ we can define the composed function $\boldsymbol{F} : \mathbb{R}^m \to \mathbb{R}^n, \boldsymbol{x} \mapsto \boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$ as follows:

$$\boldsymbol{F}(\boldsymbol{x}) = (\boldsymbol{G}_{\mathsf{outer}} \circ \boldsymbol{G}_{\mathsf{inner}})(\boldsymbol{x}) = \boldsymbol{G}_{\mathsf{outer}}(\boldsymbol{G}_{\mathsf{inner}}(\boldsymbol{x})) . \tag{15}$$

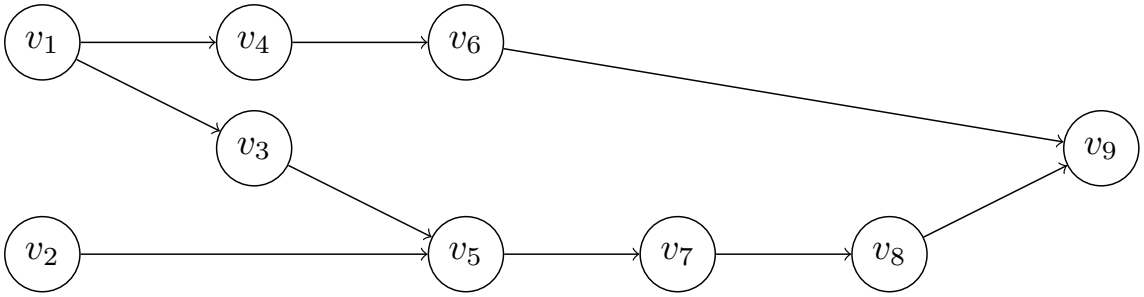


Figure 3: DAG induced by the SAC of the Rosenbrock function presented in Equation (14).

The derivative of $\boldsymbol{F}$ with respect to the input variable $\boldsymbol{x}$ can be computed using the chain rule of differential calculus

$$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{G}_{\text{outer}}}{\partial \boldsymbol{z}} \cdot \frac{\partial \boldsymbol{G}_{\text{inner}}}{\partial \boldsymbol{x}} \,. \tag{16}$$

In the context of computational differentiation, Equation (16) is applied to the SAC representation of any function $\boldsymbol{F}$ with $\boldsymbol{F}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), \ldots, f_n(\boldsymbol{x}))^T$. Therefore we need to determine the derivatives of the elemental functions φ_j defined in Equation (10) with respect to the intermediate variables $\{v_i\}_{i \prec j}$. Then, we can iteratively apply Equation (16) to compute the derivative of outputs $\boldsymbol{y} \in \mathbb{R}^n$ with respect to inputs $\boldsymbol{x} \in \mathbb{R}^m$ by the means of the elemental derivatives.

We can visualize this by augmenting the edges of the DAG by the elemental derivatives associated with the respective edge. As a result, we obtain the so-called linearized DAG [30]. The linearized DAG for Equation (14) is presented in Fig. 4.

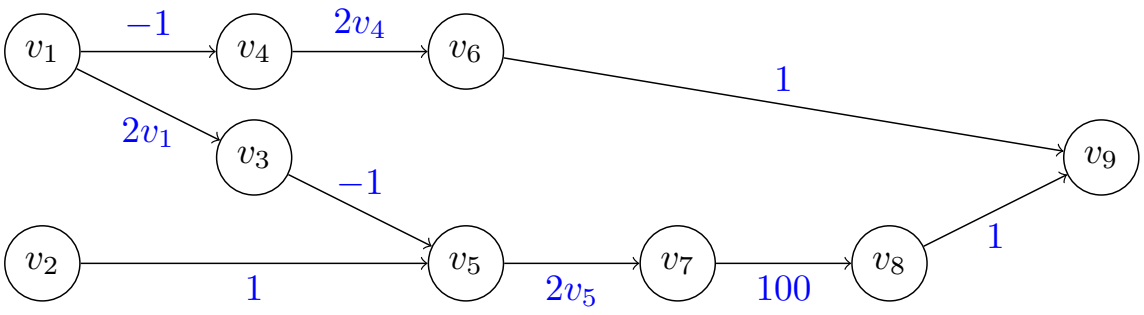


Figure 4: Linearized DAG corresponding to the SAC of the Rosenbrock function presented in Equation (14).

In order to obtain the derivative of an output variable $y_i = v_{m+p+i}$ with respect to an input variable $x_j = v_j$, we can sum over all paths P in the linearized DAG, that connect

the corresponding vertices and multiply the elemental derivatives of all edges along these paths. This yields

$$\frac{\partial y_i}{\partial x_j} = \frac{\partial f_i(\boldsymbol{x})}{\partial x_j} = \frac{\partial v_{m+p+i}}{\partial v_j} = \sum_{P \in [j \to m+p+i]} \prod_{(k,l) \in P} \frac{\partial \varphi_k}{\partial v_l} \left(\{v_q\}_{q \prec k} \right). \tag{17}$$

Here $[j \to m+p+i]$ denotes the set of all paths P in the linearized DAG which connect the input vertex j and the output vertex $m+p+i$. $(k,l) \in P$ denotes an edge contained in the path P which starts at vertex k and ends with vertex l.

With the linearized DAG depicted in Fig. 4 we can now compute the partial derivatives $\frac{\partial f_{\text{Rb}}}{\partial x_1} \equiv \frac{\partial v_9}{\partial v_1}$ and $\frac{\partial f_{\text{Rb}}}{\partial x_2} \equiv \frac{\partial v_9}{\partial v_2}$ of Equation (12) using Equation (17) the following way:

$$\begin{aligned} \frac{\partial v_9}{\partial v_1} &= \frac{\partial v_9}{\partial v_8} \cdot \frac{\partial v_8}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_3} \cdot \frac{\partial v_3}{\partial v_1} + \frac{\partial v_9}{\partial v_6} \cdot \frac{\partial v_6}{\partial v_4} \cdot \frac{\partial v_4}{\partial v_1} \\ &= -400 \cdot v_1 \cdot v_5 - 2 \cdot v_4 \end{aligned} \tag{18}$$

$$\begin{aligned} \frac{\partial v_9}{\partial v_2} &= \frac{\partial v_9}{\partial v_8} \cdot \frac{\partial v_8}{\partial v_7} \cdot \frac{\partial v_7}{\partial v_5} \cdot \frac{\partial v_5}{\partial v_2} \\ &= 200 \cdot v_5 \,. \end{aligned} \tag{19}$$

Substituting the definitions of the intermediate variables according to Equation (14) into Equations (18) and (19) yields the same results as in Equation (13).

In theory, the method presented here works even for complex functions. However, it can become very tedious to generate derivative code this way, especially as soon as the functions to be differentiated are not fully known at compile-time. Listing 4 presents an example.

Listing 4: Sample implementation of the Euclidean norm

```
double EuclideanNorm(int n, double* x) {
    double norm = 0.0;
    for(int i=0; i<n; ++i)
        norm += x[i] * x[i];
    return sqrt(norm);
}
```

In this case, the induced DAG will change at run-time depending on the input variable n. This makes the manual generation of efficient derivative code more challenging. Therefore, this work will now focus on more automated methods for generating derivatives.

2.2. Symbolic Differentiation

One of these automated methods is symbolic differentiation. In this case, the derivative is computed *symbolically* by using some software like Maple, Mathematica or Matlab. These programs take a symbolic description of the function $\boldsymbol{F}$ as input and produce an analytically exact expression for the derivative as output. Similar to the manual approach, this is realized by using the rules of differential calculus. This process is schematically visualized in Fig. 5.

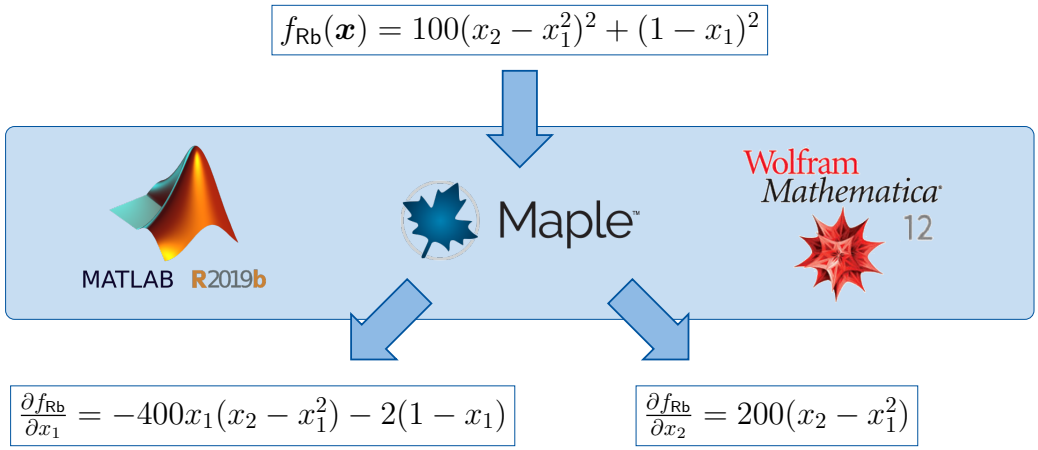


Figure 5: Schematic representation of the application of symbolic differentiation. The symbolic differentiation tool takes a symbolic description of a function as input and produces a symbolic description of the desired derivatives as output.

The result can then be evaluated in the developed software. The dependency on external software is a drawback compared to the other methods for computational differentiation as most software capable of symbolical differentiation is not freely available but has to be expensively purchased, which may not be an option for smaller projects. Additionally, the amount of time and memory needed to compute derivatives depends strongly on the complexity of the function, making symbolical differentiation inappropriate for deriving typical functions related to engineering applications.

However, external programs are not always necessary for using symbolical differentiation: Kourounis et al. [23] have implemented a symbolic differentiation framework (CoDET) in C++ using template meta-programming and expression templates. CoDET can generate partial derivatives of arbitrary order for scalar functions at compile-time, thus the simulation software only needs to evaluate the resulting derivative expression at the evaluation points. Although this method is proven to be faster than well-established automatic differentiation tools, it is still not suitable for differentiating arbitrarily complex vector-valued functions, which may result from the FEM discretization of a partial differential equation system: The compilation time would increase tremendously.

2.3. Numerical Differentiation

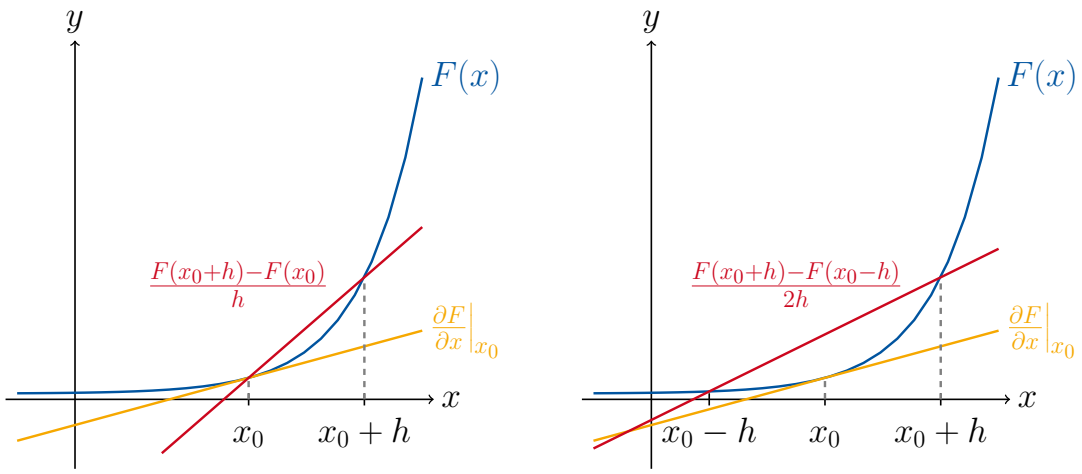
Derivatives can also be obtained *numerically* by applying a finite differencing scheme to approximate the derivative. Assume we want to determine the partial derivative of a vector valued function $\boldsymbol{F}$ with respect to some component $x_i \in \mathbb{R}, i \in \{1, ..., m\}$. Therefore we perturb the i-th component of $\boldsymbol{x} \in \mathbb{R}^m$ by some small fraction $h \in \mathbb{R}^+$ and apply a Taylor Series expansion around some point $\boldsymbol{x_0} \in \mathbb{R}^m$

$$\boldsymbol{F}(\boldsymbol{x_0} + h\, \boldsymbol{e}_i) = \boldsymbol{F}(\boldsymbol{x_0}) + \left.\frac{\partial \boldsymbol{F}}{\partial x_i}\right|_{\boldsymbol{x_0}} h + \mathcal{O}(h^2)\,, \tag{20}$$

where $\boldsymbol{e}_i \in \mathbb{R}^m$ denotes the i-th cartesian unit vector. Rearranging the terms yields an numerical expression for the wanted partial derivative

$$\left.\frac{\partial \boldsymbol{F}}{\partial x_i}\right|_{\boldsymbol{x_0}} = \frac{\boldsymbol{F}(\boldsymbol{x_0} + h\, \boldsymbol{e}_i) - \boldsymbol{F}(\boldsymbol{x_0})}{h} - \mathcal{O}(h)\,. \tag{21}$$

The forward difference approximation of a scalar function is illustrated in Fig. 6a.



(a) Forward difference approximation of a scalar function.

(b) Central difference approximation of a scalar function.

Figure 6: 1D examples for numerical differentiation using different differencing schemes. The function to be derived is shown in blue, the red line represents the finite difference approximation at point x_0 while the orange line indicates the exact analytical derivative at that point.

This numerical expression for the partial derivatives can now be used to compute more complex differential expressions. The partial derivative $\frac{\partial \boldsymbol{F}(\boldsymbol{x})}{\partial x_i}$ corresponds to the i-th column of the Jacobian $\mathbf{J}_{\boldsymbol{F}} \in \mathbb{R}^{n \times m}$ of $\boldsymbol{F}$. Therefore, we only need to vary $\boldsymbol{e}_i$ over all cartesian unit vectors ($i = 1, \ldots, m$) to get the full Jacobian at a run-time cost of $\mathcal{O}(m) \cdot \mathrm{cost}(\boldsymbol{F})$. The advantage of the numerical approach is that it is generally easy to

implement this method into (already existing) simulation programs, as it only requires additional evaluations of the function to be derived at some new points. Although this strategy can be widely applied, it has some drawbacks. Due to the truncation error introduced by the spatial discretization, computing derivatives this way is no longer analytically exact. For the presented one-sided forward differences, the error of the numerical approximation compared to the exact derivative scales linearly with the step width h. Lowering the truncation error (that is, increasing the accuracy of the finite differencing scheme) requires additional function evaluations [14]. E.g., by using central differences

$$\left.\frac{\partial \boldsymbol{F}}{\partial x_i}\right|_{\boldsymbol{x_0}} = \frac{\boldsymbol{F}(\boldsymbol{x_0} + h\,\boldsymbol{e}_i) - \boldsymbol{F}(\boldsymbol{x_0} - h\,\boldsymbol{e}_i)}{2h} - \mathcal{O}(h^2)\,, \tag{22}$$

one can reduce the truncation error by one power, which may lead to better approximations, as depicted in Fig. 6b. However, it results in a linear increase of the run-time for computing the derivative. Another caveat is the choice of the step width h. This is not trivial and depends on the individual application.

2.4. Algorithmic Differentiation

The term *algorithmic differentiation* comprises a variety of techniques, which enable the computer to generate desired derivative values from a provided function implementation [22]. Although the run-time for obtaining derivatives using some kind of algorithmic differentiation technique typically is still higher compared to a manual implementation, they overcome the accuracy drawback of numerical differentiation: AD strategies do not incorporate any truncation error and are thus exact with respect to machine precision. There are different ways how code for the derivatives can be generated: Juedes [22] presents a taxonomy of automatic differentiation tools – mainly for programs written in FORTRAN –, which aims to classify some AD tools based on the level of integration into the programming language. Nowadays, we typically distinguish between two different implementation strategies, namely automatic differentiation by source code transformation and automatic differentiation using operator overloading. Both are further explained in the following sections. Besides the implementational aspect, there also exist different mathematical approaches for obtaining derivatives. The fundamentals of forward mode and backward mode differentiation are presented afterwards. More profound information can be found in the books of Naumann [30] and Griewank and Walther [18].

2.4.1. Source Code Transformation

One possibility to implement AD is using *source code transformation*. Thereby, the source code containing the function $\boldsymbol{F}$ is parsed and a new source file is generated, which implements the desired derivative. The flow of information is visualized in Fig. 7.

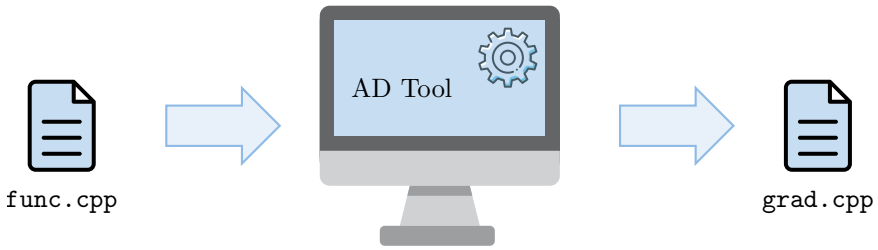


Figure 7: Schematic representation of AD via source code transformation. A source file implementing the function is passed to the AD tool which parses the function and generates a new source file containing the corresponding derivative code.

This technique has been mainly applied to scientific codes written in FORTRAN, e.g., see the tools `ADIFOR` by Bischof et al. [5] or `OpenAD/F` by Utke et al. [38]. There is only very little support for C/C++ codes. Examples are `ADIC2` by Narayanan et al. [29] or `Tapenade` by Hascoet and Pascual [20], which was originally developed to transform FORTRAN code but has later been extended to some features of C/C++. Although there are some software implementations available for C++, these are generally very restrictive in terms of the supported language features and none of them can deal with modern templatized C++17 codes.

2.4.2. Operator Overloading

Implementing AD by operator overloading is a straight forward approach in most object-oriented programming languages such as C++. The key idea is to create a new datatype with specialized arithmetical operators and mathematical functions that do not only evaluate the elemental function but compute the associated elemental derivative as well. There are a lot of software libraries available, providing such data types. The first libraries have already been released in the last century, including the packages `ADOL-C` by Griewank et al. [19] and `FADBAD` by Bendtsen and Stauning [4]. Since then, various efforts have been made to improve the efficiency of the various AD implementations. Margossian [26] gives an overview of several optimization techniques for efficient implementation, including retaping, checkpointing, the usage of region-based memory, or expression templates. The latter ones have already been used for implementing forward mode AD by Aubert et al. [2] and for the reverse mode implementation of the `ADEPT` library by Hogan [21]. Based on the idea of expression

templates, Phipps and Pawlowski [33] developed a special caching strategy in order to reduce the amount of duplicated computations, which are usually related to especially large expression trees. This strategy has been applied to compute the element-level Jacobian matrices for the nonlinear FEM discretization of a complex fluid dynamics problem. Such a hybrid approach that uses AD to compute the derivatives for a relatively small subset of functions on an element-level basis has also been examined by Bartlett et al. [3]. The obtained element-level matrices are afterwards broadcast manually into the global derivative matrix corresponding to the full system. The application of an AD tool to a complex simulation software is also discussed by Sagebaum et al. [35], where some strategies for a transition from an existing simulation code with manual differentiation to one with automatic differentiation capabilities are described. Some of these considerations will influence the development of the CD framework described in Chapter 3.

The next sections present two mathematical algorithms which can be used to obtain derivatives algorithmically.

2.4.3. Forward Mode

When using the *forward* or *tangent* mode of AD, the projection of the derivative of the multivariate vector valued function $\boldsymbol{F} : \mathbb{R}^m \to \mathbb{R}^n$ onto a given directional vector $\boldsymbol{x}^{(\mathrm{t})} \in \mathbb{R}^m$ is considered. For this projection, we can define the following mapping $\boldsymbol{F}^{(\mathrm{t})}$ which is induced by the function's Jacobian $\mathbf{J}_{\boldsymbol{F}}$:

$$\boldsymbol{F}^{(\mathrm{t})} : \mathbb{R}^{2m} \to \mathbb{R}^n, \left(\boldsymbol{x}, \boldsymbol{x}^{(\mathrm{t})}\right) \mapsto \boldsymbol{y}^{(\mathrm{t})} = \boldsymbol{F}^{(\mathrm{t})}\left(\boldsymbol{x}, \boldsymbol{x}^{(\mathrm{t})}\right) = \mathbf{J}_{\boldsymbol{F}}(\boldsymbol{x}) \cdot \boldsymbol{x}^{(\mathrm{t})} . \tag{23}$$

The result of the mapping can be regarded as the directional derivative of $\boldsymbol{F}$ with respect to the direction of $\boldsymbol{x}^{(\mathrm{t})}$. Thus, one can obtain the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$ by letting $\boldsymbol{x}^{(\mathrm{t})}$ range over all cartesian unit vectors $\boldsymbol{e}_j \in \mathbb{R}^m$ with $j = 1, \ldots, m$. The multiplication with a distinct unit vector $\boldsymbol{e}_j$ yields one column of the Jacobian as visualized in Fig. 8.

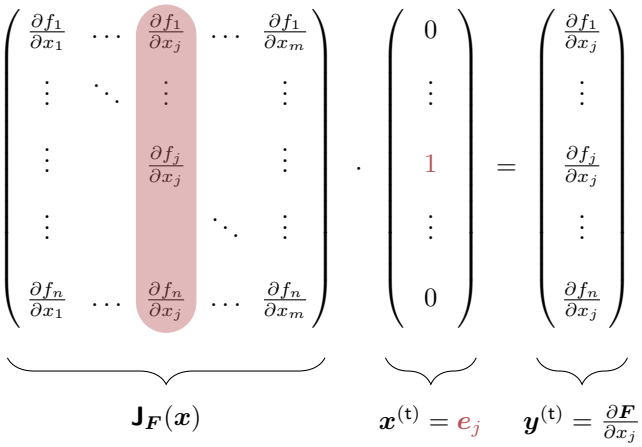


Figure 8: By setting the input vector $\boldsymbol{x}^{(\mathrm{t})}$ to a distinct unit vector $\boldsymbol{e}_j$, forward mode differentiation can compute one column of the Jacobian $\mathbf{J}_{\boldsymbol{F}}$.

The mapping $\boldsymbol{F}^{(\mathrm{t})}$ can directly be applied to the SAC representation of a function as described in Equation (10). This yields the following algorithm:

Algorithm 1 Forward Mode Differentiation

Input: $\boldsymbol{x}$, $\boldsymbol{x}^{(\mathrm{t})}$
Result: $\boldsymbol{y}$, $\boldsymbol{y}^{(\mathrm{t})}$
Start
$(v_1, \ldots, v_m) \leftarrow \boldsymbol{x}$ ▷ Copy states
$(v_1^{(\mathrm{t})}, \ldots, v_m^{(\mathrm{t})}) \leftarrow \boldsymbol{x}^{(\mathrm{t})}$ ▷ Copy directional vector
for $j = m+1, \ldots, m+p+n$ **do**
  $v_j \leftarrow \varphi_j(\{v_i\}_{i \prec j})$ ▷ Evaluate function
  $v_j^{(\mathrm{t})} \leftarrow \sum_{i:\, i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_i^{(\mathrm{t})}$ ▷ Evaluate derivative
end for
$\boldsymbol{y} \leftarrow (v_{m+p+1}, \ldots, v_{m+p+n})$ ▷ Extract function value
$\boldsymbol{y}^{(\mathrm{t})} \leftarrow (v_{m+p+1}^{(\mathrm{t})}, \ldots, v_{m+p+n}^{(\mathrm{t})})$ ▷ Extract directional derivative
End

Table 1 shows the application of Algorithm 1 to the example of the Rosenbrock function Equation (12). The results correspond to the analytical solution, which can be obtained by evaluating Equation (12) and Equation (13). In this example, only the partial derivative with respect to the first input x_1 has been computed. As mentioned before, the computation of the full Jacobian requires a total number of m evaluations of $\boldsymbol{F}^{(\mathrm{t})}$, which results in a linear run-time proportional to $\mathcal{O}(m) \cdot \mathrm{cost}(\boldsymbol{F}^{(\mathrm{t})})$. Thus, the run-time of tangent mode AD is similar to that of a finite difference approximation but AD is more accurate as no discretization error is introduced. However, m may be very large depending on the application (e.g., the number of unknowns of an FEM dis-

$v_1 =$	x_1	$= -1$
$v_2 =$	x_2	$= 1$
$v_1^{(\mathsf{t})} =$	$x_1^{(\mathsf{t})}$	$= 1$
$v_2^{(\mathsf{t})} =$	$x_2^{(\mathsf{t})}$	$= 0$
$v_3 =$	v_1^2	$= 1$
$v_3^{(\mathsf{t})} =$	$2 \cdot v_1 \cdot v_1^{(\mathsf{t})}$	$= -2$
$v_4 =$	$1 - v_1$	$= 2$
$v_4^{(\mathsf{t})} =$	$(-1) \cdot v_1^{(\mathsf{t})}$	$= -1$
$v_5 =$	$v_2 - v_3$	$= 0$
$v_5^{(\mathsf{t})} =$	$1 \cdot v_2^{(\mathsf{t})} + (-1) \cdot v_3^{(\mathsf{t})}$	$= 2$
$v_6 =$	v_4^2	$= 4$
$v_6^{(\mathsf{t})} =$	$2 \cdot v_4 \cdot v_4^{(\mathsf{t})}$	$= -4$
$v_7 =$	v_5^2	$= 0$
$v_7^{(\mathsf{t})} =$	$2 \cdot v_5 \cdot v_5^{(\mathsf{t})}$	$= 0$
$v_8 =$	$100 \cdot v_7$	$= 0$
$v_8^{(\mathsf{t})} =$	$100 \cdot v_7^{(\mathsf{t})}$	$= 0$
$v_9 =$	$v_8 + v_6$	$= 4$
$v_9^{(\mathsf{t})} =$	$1 \cdot v_8^{(\mathsf{t})} + 1 \cdot v_6^{(\mathsf{t})}$	$= -4$
$y =$	v_9	$= 4$
$y^{(\mathsf{t})} =$	$v_9^{(\mathsf{t})}$	$= -4$

Table 1: Algorithm 1 applied to the Rosenbrock function (Equation (12)) to compute its derivative at point $\boldsymbol{x} = (-1, 1)^T$ in the direction of $\boldsymbol{x}^{(\mathsf{t})} = (1, 0)^T$.

cretization), resulting in possibly very long run-times. Here, the reverse mode of AD can often yield a better performance.

2.4.4. Reverse Mode

Reverse or *adjoint* mode of AD can be used to compute the derivative of a function with a run-time that scales linearly with the number of outputs instead of the number of inputs, resulting in a total run-time of $\mathcal{O}(n) \cdot \text{cost}(\boldsymbol{F}^{(\mathsf{a})})$. Here, $\boldsymbol{F}^{(\mathsf{a})}$ is another mapping, which is also induced by the function's Jacobian $\mathbf{J}_{\boldsymbol{F}}$:

$$\boldsymbol{F}^{(\mathsf{a})} : \mathbb{R}^{m+n} \to \mathbb{R}^m, \left(\boldsymbol{x}, \boldsymbol{y}^{(\mathsf{a})}\right) \mapsto \boldsymbol{x}^{(\mathsf{a})} = \boldsymbol{F}^{(\mathsf{a})}\left(\boldsymbol{x}, \boldsymbol{y}^{(\mathsf{a})}\right) = \mathbf{J}_{\boldsymbol{F}}^T(\boldsymbol{x}) \cdot \boldsymbol{y}^{(\mathsf{a})} . \tag{24}$$

Similar to the case of forward mode AD, the whole Jacobian can be computed by letting $\boldsymbol{y}^{(\mathsf{a})}$ range over all unit vectors $\boldsymbol{e}_i$ with $i = 1, \ldots, n$. For each unit vector $\boldsymbol{e}_i$ one obtains one row of $\mathbf{J}_{\boldsymbol{F}}$. This is illustrated in Fig. 9.

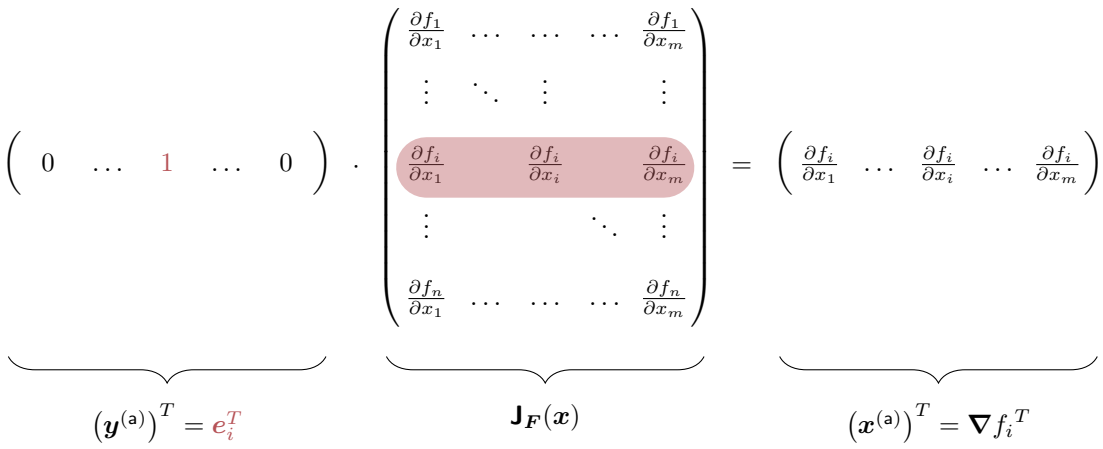


Figure 9: By setting the input vector $\boldsymbol{y}^{(\mathsf{a})}$ to a distinct unit vector $\boldsymbol{e}_i$, reverse mode differentiation can compute one row of the Jacobian $\mathbf{J}_{\boldsymbol{F}}$.

As implied by its name, the adjoints are propagated backwards through the SAC. This requires two steps: First, all elemental functions need to be evaluated. In contrast to forward mode differentiation, all intermediate values need to be stored, which means that this method requires additional memory. With the stored auxiliary values, the adjoints are then computed in a succeeding iteration. This can be summarized in the following algorithm:

Algorithm 2 Reverse Mode Differentiation

Input: $\boldsymbol{x}$, $\boldsymbol{y}^{(\mathsf{a})}$
Result: $\boldsymbol{y}$, $\boldsymbol{x}^{(\mathsf{a})}$
Start
  $(v_1, \dots, v_m) \leftarrow \boldsymbol{x}$ ▷ Copy state
  $(v_{m+p+1}^{(\mathsf{a})}, \dots, v_{m+p+n}^{(\mathsf{a})}) \leftarrow \boldsymbol{y}^{(\mathsf{a})}$ ▷ Copy adjoint vector
  for $j = m+1, \dots, m+p+n$ **do**
    $v_j \leftarrow \varphi_j(\{v_i\}_{i \prec j})$ ▷ Evaluate function
  end for
  for $i = n+p, \dots, 1$ **do**
    $v_i^{(\mathsf{a})} \leftarrow \sum_{j:\, i \prec j} \frac{\partial \varphi_j}{\partial v_i} \cdot v_j^{(\mathsf{a})}$ ▷ Evaluate adjoints
  end for
  $\boldsymbol{y} \leftarrow (v_{m+p+1}, \dots, v_{m+p+n})$ ▷ Extract function values
  $\boldsymbol{x}^{(\mathsf{a})} \leftarrow (v_1^{(\mathsf{a})}, \dots, v_m^{(\mathsf{a})})$ ▷ Extract adjoints
End

This algorithm can be applied to compute the full gradient of the Rosenbrock function with only one call. The result is shown in Table 2.

$v_1 =$	x_1	$= -1$
$v_2 =$	x_2	$= 1$
$v_9^{(\mathsf{a})} =$	$y^{(\mathsf{a})}$	$= 1$
$v_3 =$	v_1^2	$= 1$
$v_4 =$	$1 - v_1$	$= 2$
$v_5 =$	$v_2 - v_3$	$= 0$
$v_6 =$	v_4^2	$= 4$
$v_7 =$	v_5^2	$= 0$
$v_8 =$	$100 \cdot v_7$	$= 0$
$v_9 =$	$v_8 + v_6$	$= 4$
$v_8^{(\mathsf{a})} =$	$1 \cdot v_9^{(\mathsf{a})}$	$= 1$
$v_7^{(\mathsf{a})} =$	$100 \cdot v_8^{(\mathsf{a})}$	$= 100$
$v_6^{(\mathsf{a})} =$	$1 \cdot v_9^{(\mathsf{a})}$	$= 1$
$v_5^{(\mathsf{a})} =$	$2 \cdot v_5 \cdot v_7^{(\mathsf{a})}$	$= 0$
$v_4^{(\mathsf{a})} =$	$2 \cdot v_4 \cdot v_6^{(\mathsf{a})}$	$= 4$
$v_3^{(\mathsf{a})} =$	$(-1) \cdot v_5^{(\mathsf{a})}$	$= 0$
$v_2^{(\mathsf{a})} =$	$1 \cdot v_5^{(\mathsf{a})}$	$= 0$
$v_1^{(\mathsf{a})} =$	$2 \cdot v_1 \cdot v_3^{(\mathsf{a})} + (-1) \cdot v_4^{(\mathsf{a})}$	$= -4$
$y =$	v_9	$= 4$
$x_1^{(\mathsf{a})} =$	$v_1^{(\mathsf{a})}$	$= -4$
$x_2^{(\mathsf{a})} =$	$v_2^{(\mathsf{a})}$	$= 0$

Table 2: Algorithm 2 applied to the Rosenbrock function (Equation (12)) to compute the full gradient at point $\boldsymbol{x} = (-1, 1)^T$ for the input $y^{(\mathsf{a})} = 1$.

As mentioned in the previous sections, there are numerous software packages available, which provide various efficient AD implementations. Many can be freely downloaded and integrated into one's own software projects. An overview of current software packages can be found at `http://www.autodiff.org/`.

However, in this thesis, we will implement our own CD framework and tailor it to the architecture of CAMPIGA. Regarding AD, forward mode differentiation using operator overloading has been chosen for the following reasons: In CAMPIGA we want to use CD to determine the contribution of single elements to the Jacobian of a nonlinear equation system resulting from an FEM discretization of a PDE system. On the element level, we always have the same number of unknowns and equations, so that for our case $m = n$ holds. Therefore, reverse mode differentiation is no longer guaranteed to be faster, while simultaneously being more memory-intensive and harder to implement. The focus will be on the implementation of a simple datatype for forward mode differentiation, which can support the early stages of the development process as it allows to rapidly extend CAMPIGA by newly implemented equations or mate-

rial models without the overhead of providing an additional implementation of the required derivative.

3. Computational Differentiation in CAMPIGA

This chapter explains the development of a CD framework for CAMPIGA in detail. Section 3.1 contains the mathematical foundations of the FEM approach used to solve PDE systems within CAMPIGA. The discretization of the PDEs will lead to a nonlinear equation system, which is solved by applying Newton's method. The required Jacobian is supposed to be computed by means of CD. After that the integration of forward mode differentiation as well as finite differences into the existing CAMPIGA code is described in the Sections 3.2 to 3.5. Specialized data types are constructed to compute the derivative of an expression together with its value. An example is provided to give an insight on their functionality. Finally, the required adaptions to fully integrate the new data types are described.

3.1. Stationary Model Equations

As mentioned in the introduction, CAMPIGA is a simulation code that aims to solve problems arising in engineering applications using FEM. For this thesis, we want to focus on the solution of fluid mechanics problems. The example problems, which we want to examine in Chapter 4, are all governed by the stationary, incompressible Navier-Stokes Equations (NSE), which read in an Eulerian frame of reference:

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0 \qquad \text{in } \Omega \tag{25}$$

$$\rho\, \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \mathbf{0} \qquad \text{in } \Omega \tag{26}$$

$$\boldsymbol{v} = \boldsymbol{v}_{\mathsf{D}} \qquad \text{on } \Gamma_{\mathsf{D}} \tag{27}$$

$$\boldsymbol{n} \cdot \boldsymbol{\sigma} = \boldsymbol{t}_{\mathsf{N}} \qquad \text{on } \Gamma_{\mathsf{N}}\,. \tag{28}$$

Here, $\boldsymbol{v}$ denotes the velocity vector, ρ corresponds to the fluid's density and $\boldsymbol{\sigma}$ is the Cauchy stress tensor. $\boldsymbol{v}_{\mathsf{D}}$ denotes the velocity on the Dirichlet part of the boundary, while $\boldsymbol{t}_{\mathsf{N}}$ is the prescribed traction on the Neumann part of the boundary. The PDE system requires an additional closure equation for the Cauchy stress. Within this thesis, we only pay attention to Newtonian fluids. This means, that the stress tensor takes on the following form

$$\boldsymbol{\sigma}(p, \boldsymbol{v}) = \eta \left(\boldsymbol{\nabla} \boldsymbol{v}^{\mathsf{T}} + \boldsymbol{\nabla} \boldsymbol{v} \right) - p\, \mathbf{Id}\,, \tag{29}$$

where η is the dynamic viscosity and p denotes the pressure. **Id** symbolizes the identity matrix.

In order to solve these equations using an FEM approach, we first need to derive a weak formulation. The required function spaces are given by:

$$\mathcal{S}_p = \mathcal{L}^2(\Omega) \tag{30}$$

$$\mathcal{S}_{\boldsymbol{v}} = \left\{ \boldsymbol{v} \in (H^1(\Omega))^{n_{\text{sd}}} \;\middle|\; \boldsymbol{v} = \boldsymbol{v}_{\text{D}} \text{ on } \Gamma_{\text{D}} \right\} \tag{31}$$

$$\mathcal{V}_{\text{C}} = \mathcal{L}^2(\Omega) \tag{32}$$

$$\mathcal{V}_{\text{M}} = \left\{ w \in H^1(\Omega) \;\middle|\; w = 0 \text{ on } \Gamma_{\text{D}} \right\} . \tag{33}$$

Using these function spaces, we can now multiply Equations (25) and (26) with test functions $w_{\text{C}} \in \mathcal{V}_{\text{C}}$ and $w_{\text{M}} \in \mathcal{V}_{\text{M}}$ and integrate them over the computational domain Ω. This yields the following weak formulation:

Find $(p, \boldsymbol{v}) \in \mathcal{S}_p \times \mathcal{S}_{\boldsymbol{v}}$ such that

$$\int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{v})\, w_{\text{C}} \, \mathrm{d}\Omega = 0 \tag{34}$$

$$\int_\Omega (\rho\, \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma}(p, \boldsymbol{v}))\, w_{\text{M}} \, \mathrm{d}\Omega = \boldsymbol{0} \,, \tag{35}$$

for all $(w_{\text{C}}, w_{\text{M}}) \in \mathcal{V}_{\text{C}} \times \mathcal{V}_{\text{M}}$.

The only unknowns in the equation system above are the velocity $\boldsymbol{v}$ and the pressure p, all other quantities are parameters, which characterize the fluid. Integrating the Cauchy stress by parts and using the fact that by construction $w_{\text{M}} = 0$ on Γ_{D}, we can rewrite the weak formulation as follows:

Find $(p, \boldsymbol{v}) \in \mathcal{S}_p \times \mathcal{S}_{\boldsymbol{v}}$ such that

$$\int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{v})\, w_{\text{C}} \, \mathrm{d}\Omega = 0 \tag{36}$$

$$\int_\Omega \rho\, (\boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v})\, w_{\text{M}} \, \mathrm{d}\Omega + \int_\Omega \boldsymbol{\sigma}(p, \boldsymbol{v}) \cdot \boldsymbol{\nabla} w_{\text{M}} \, \mathrm{d}\Omega = \boldsymbol{0} \,, \tag{37}$$

for all $(w_{\text{C}}, w_{\text{M}}) \in \mathcal{V}_{\text{C}} \times \mathcal{V}_{\text{M}}$.

For the FEM discretization, we want to solve this weak formulation on a computational grid Ω^{h}. We are only interested in the unknown quantities on the discrete grid points. Therefore, we select an ansatz for the pressure p and the velocity $\boldsymbol{v}$, which is defined by the nodal values of the unknowns. Let η_p and η_v denote the sets of all nodes associated with the pressure and the velocity field. $\eta_{\text{D},v}$ is the part of the velocity nodes, which lies on the Dirichlet boundary Γ_{D}. We then sum over all nodes and multiply the nodal value of the unknowns at each node with a properly chosen ansatz function. This yields

$$p^{\text{h}}(\boldsymbol{x}) = \sum_{a \in \eta_p} \hat{p}^a \phi_p^a(\boldsymbol{x}) \tag{38}$$

$$\boldsymbol{v}^{\text{h}}(\boldsymbol{x}) = \underbrace{\sum_{i=1}^{n_{\text{sd}}} \sum_{b \in \eta_v \setminus \eta_{\text{D}_i,v}} \hat{v}_i^b \, \phi_v^b(\boldsymbol{x}) \, \boldsymbol{e}_i}_{\tilde{\boldsymbol{v}}^{\text{h}}} + \underbrace{\sum_{i=1}^{n_{\text{sd}}} \sum_{b \in \eta_{\text{D}_i,v}} v_{\text{D}_i}(\boldsymbol{x}) \, \phi_v^b(\boldsymbol{x}) \, \boldsymbol{e}_i}_{\boldsymbol{v}_{\text{D}}^{\text{h}}} . \tag{39}$$

Here, ϕ_p^a and ϕ_v^b denote the ansatz functions for the pressure and a velocity component on a given pressure node a and velocity node b, while $\hat{p}^a$ and $\hat{v}_i^b$ represent the pressure value and the i-th velocity component at these nodes. n_{sd} stands for the number of spatial dimensions where this thesis is restricted to two. The approximation for the velocity field has been split up into a function $\boldsymbol{v}_{\mathsf{D}}^{\mathsf{h}}$, which is completely defined by the Dirichlet boundary conditions, and a function $\tilde{\boldsymbol{v}}^{\mathsf{h}}$ that needs to be computed.
There are several possible choices for ansatz functions. In the following we will assume that $\phi_p^a, \phi_v^b \in \mathcal{P}^q(\Omega^e)$ holds, where $\mathcal{P}^q(\Omega_{\mathsf{ref}})$ is the space of polynomials of order q on the reference element Ω_{ref}. The ansatz functions are constructed such, that they take on a value of 1 only on the node with the global node number a and are zero on every other node.

In order to obtain a discretized weak formulation we also need discrete ansatz functions to test the continuity and momentum equation. For the standard Galerkin FEM these discrete test functions are defined as:

$$\phi_{\mathsf{C}}^A = \phi_p^a \tag{40}$$
$$\phi_{\mathsf{M}}^B = \phi_v^a \,. \tag{41}$$

With the help of these discrete ansatz functions, we can now define the discrete solution spaces as follows:

$$\mathcal{S}_p^{\mathsf{h}} = \operatorname*{span}_{a\in\eta_p}\{\phi_p^a\} \tag{42}$$
$$\mathcal{S}_{\boldsymbol{v}}^{\mathsf{h}} = \left\{ \tilde{\boldsymbol{v}}^{\mathsf{h}} \;\middle|\; \tilde{v}_i^{\mathsf{h}} \in \operatorname*{span}_{b\in\eta_v\setminus\eta_{\mathsf{D}_i,v}} \{\phi_v^b\} \,,\; i = 1,\ldots,n_{\mathsf{sd}} \right\} \tag{43}$$
$$\mathcal{V}_{\mathsf{C}}^{\mathsf{h}} = \operatorname*{span}_{a\in\eta_p}\{\phi_{\mathsf{C}}^A\} \tag{44}$$
$$\mathcal{V}_{\mathsf{M}}^{\mathsf{h}} = \operatorname*{span}_{a\in\eta_v}\{\phi_{\mathsf{M}}^B\} \,. \tag{45}$$

The discretized weak formulation now reads:

Find $(p^{\mathsf{h}}, \tilde{\boldsymbol{v}}^{\mathsf{h}}) \in \mathcal{S}_p^{\mathsf{h}} \times \mathcal{S}_{\boldsymbol{v}}^{\mathsf{h}}$ such that

$$\int_\Omega (\boldsymbol{\nabla}\cdot\boldsymbol{v}^{\mathsf{h}})\,\phi_{\mathsf{C}}^A \,\mathrm{d}\Omega = 0 \tag{46}$$
$$\int_\Omega \rho\,(\boldsymbol{v}^{\mathsf{h}}\cdot\boldsymbol{\nabla}\boldsymbol{v}^{\mathsf{h}})\,\phi_{\mathsf{M}}^B \,\mathrm{d}\Omega + \int_\Omega \boldsymbol{\sigma}(p^{\mathsf{h}},\boldsymbol{v}^{\mathsf{h}})\cdot\boldsymbol{\nabla}\phi_{\mathsf{M}}^B \,\mathrm{d}\Omega = \boldsymbol{0}\,, \tag{47}$$

for all $(\phi_{\mathsf{C}}^A, \phi_{\mathsf{M}}^B) \in \mathcal{V}_{\mathsf{C}}^{\mathsf{h}} \times \mathcal{V}_{\mathsf{M}}^{\mathsf{h}}$.

The discretized weak formulation is essentially a nonlinear equation system, where the nonlinearity stems from the convective part of the NSE. The global equation system has the form

$$\boldsymbol{R}(\boldsymbol{u}) = \begin{pmatrix} C^1(\boldsymbol{u}) \\ \vdots \\ C^{n_{\mathsf{n},p}}(\boldsymbol{u}) \\ \boldsymbol{M}^1(\boldsymbol{u}) \\ \vdots \\ \boldsymbol{M}^{n_{\mathsf{n},v}}(\boldsymbol{u}) \end{pmatrix} = \boldsymbol{0}\,. \tag{48}$$

The component functions are defined according to the discretized weak formulation

$$C^A(\boldsymbol{u}) = \int_\Omega (\boldsymbol{\nabla}\cdot\boldsymbol{v}^{\mathsf{h}})\,\phi^A_{\mathsf{C}}\,\mathrm{d}\Omega \tag{49}$$

$$\boldsymbol{M}^B(\boldsymbol{u}) = \int_\Omega \rho\,(\boldsymbol{v}^{\mathsf{h}}\cdot\boldsymbol{\nabla}\boldsymbol{v}^{\mathsf{h}})\,\phi^B_{\mathsf{M}}\,\mathrm{d}\Omega + \int_\Omega \boldsymbol{\sigma}(p^{\mathsf{h}},\boldsymbol{v}^{\mathsf{h}})\cdot\boldsymbol{\nabla}\phi^B_{\mathsf{M}}\,\mathrm{d}\Omega\,, \tag{50}$$

where $A = 1,\ldots,n_{\mathsf{n},p}$ and $B = 1,\ldots,n_{\mathsf{n},v}$. $n_{\mathsf{n},p} = |\eta_p|$ denotes the total number of pressure nodes and $n_{\mathsf{n},v} = |\eta_v \setminus \eta_{\mathsf{D},v}|$ the total number of velocity nodes accordingly. The unknowns are the nodal pressure and velocity values $\hat{p}^a$ and $\hat{v}^b_i$. They are grouped together in the global vector of unknowns $\boldsymbol{u}$

$$\boldsymbol{u} = (\hat{p}^1,\ldots,\hat{p}^{n_{\mathsf{n},p}},\hat{v}^1_1,\ldots,\hat{v}^1_{n_{\mathsf{sd}}},\hat{v}^2_1,\ldots,\hat{v}^{n_{\mathsf{n},v}}_{n_{\mathsf{sd}}})^{\mathsf{T}}\,. \tag{51}$$

This nonlinear equation system can be solved iteratively using a Newton-Raphson method. In order to compute the next iterand $\boldsymbol{u}^{\nu+1}$, a Taylor series expansion of $\boldsymbol{R}(\boldsymbol{u})$ around the previous iterand $\boldsymbol{u}^\nu$ is applied and truncated after the linear term. This yields:

$$\boldsymbol{R}(\boldsymbol{u}^{\nu+1}) = \boldsymbol{R}(\boldsymbol{u}^\nu) + \mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu)\cdot(\boldsymbol{u}^{\nu+1}-\boldsymbol{u}^\nu) = \boldsymbol{0}\,. \tag{52}$$

Rearranging the equations results in a linear system, which needs to be solved in each Newton step:

$$\mathbf{J}_{\boldsymbol{R}}(\boldsymbol{u}^\nu)\cdot\Delta\boldsymbol{u}^\nu = -\boldsymbol{R}(\boldsymbol{u}^\nu) \tag{53}$$

$$\boldsymbol{u}^{\nu+1} = \boldsymbol{u}^\nu + \Delta\boldsymbol{u}^\nu\,. \tag{54}$$

Usually, we do not assemble Equation (48) and its Jacobian globally, but locally on an element-level basis. For the simulation, the computational domain Ω is discretized by a mesh consisting of n_{el} elements Ω^e. Instead of integrating over the whole domain Ω we can integrate the weak formulation on each element and sum up all contributions:

$$C^A(\boldsymbol{u}) = \sum_{e=1}^{n_{\mathsf{el}}}\int_{\Omega^e}(\boldsymbol{\nabla}\cdot\boldsymbol{v}^{\mathsf{h}})\,\phi^A_{\mathsf{C}}\,\mathrm{d}\Omega^e \tag{55}$$

$$\boldsymbol{M}^B(\boldsymbol{u}) = \sum_{e=1}^{n_{\mathsf{el}}}\int_{\Omega^e}\rho\,(\boldsymbol{v}^{\mathsf{h}}\cdot\boldsymbol{\nabla}\boldsymbol{v}^{\mathsf{h}})\,\phi^B_{\mathsf{M}}\,\mathrm{d}\Omega^e + \int_{\Omega^e}\boldsymbol{\sigma}(p^{\mathsf{h}},\boldsymbol{v}^{\mathsf{h}})\cdot\boldsymbol{\nabla}\phi^B_{\mathsf{M}}\,\mathrm{d}\Omega^e\,. \tag{56}$$

We can simplify this formulation further by looking at the definitions of $\boldsymbol{v}^{\mathsf{h}}$ and p^{h} in Equations (38) and (39). Due to the choice of ansatz functions, we can neglect all contributions associated with nodes that are not part of the current element. Thus it suffices to iterate over the nodes on the current element. We write

$$p^{\mathsf{h},e}(\boldsymbol{x}) = \sum_{n=1}^{n_{\mathsf{en},p}} \hat{p}^n \; \phi_p^n(\boldsymbol{x}) \tag{57}$$

$$\boldsymbol{v}^{\mathsf{h},e}(\boldsymbol{x}) = \sum_{i=1}^{n_{\mathsf{sd}}} \sum_{n=1}^{n_{\mathsf{en},v}} \hat{v}_i^n \; \phi_v^n(\boldsymbol{x}) \; \boldsymbol{e}_i \,. \tag{58}$$

where $n_{\mathsf{en},p}$ and $n_{\mathsf{en},v}$ denote the number of element nodes for pressure and velocity respectively. With this information we can rewrite Equations (55) and (56):

$$C^A(\boldsymbol{u}) = \sum_{e=1}^{n_{\mathsf{el}}} C^{A,e}(\boldsymbol{u}^e) \tag{59}$$

$$\boldsymbol{M}^B(\boldsymbol{u}) = \sum_{e=1}^{n_{\mathsf{el}}} \boldsymbol{M}^{B,e}(\boldsymbol{u}^e) \,, \tag{60}$$

with

$$C^{A,e}(\boldsymbol{u}^e) = \int_{\Omega^e} (\boldsymbol{\nabla} \cdot \boldsymbol{v}^{\mathsf{h},e}) \; \phi_{\mathsf{C}}^A \; \mathrm{d}\Omega^e \tag{61}$$

$$\boldsymbol{M}^{B,e}(\boldsymbol{u}^e) = \int_{\Omega^e} \rho \; (\boldsymbol{v}^{\mathsf{h},e} \cdot \boldsymbol{\nabla} \boldsymbol{v}^{\mathsf{h},e}) \; \phi_{\mathsf{M}}^B \; \mathrm{d}\Omega^e + \int_{\Omega^e} \boldsymbol{\sigma}(p^{\mathsf{h},e}, \boldsymbol{v}^{\mathsf{h},e}) \cdot \boldsymbol{\nabla} \phi_{\mathsf{M}}^B \; \mathrm{d}\Omega^e \,. \tag{62}$$

The vector $\boldsymbol{u}^e$ contains the element-level local unknowns:

$$\boldsymbol{u}^e = (\hat{p}^1, \ldots, \hat{p}^{n_{\mathsf{en},p}}, \hat{v}_1^1, \ldots, \hat{v}_{n_{\mathsf{sd}}}^1, \hat{v}_1^2, \ldots, \hat{v}_{n_{\mathsf{sd}}}^{n_{\mathsf{en},v}})^{\mathsf{T}} \,. \tag{63}$$

As the test functions stem from similar function spaces as the ansatz functions, we can analogously argue that Equations (61) and (62) only need to be tested with test functions associated with nodes on the current element. This leads to the following definition of the element-level residual:

$$\boldsymbol{R}^e(\boldsymbol{u}^e) = \begin{pmatrix} C^{1,e}(\boldsymbol{u}^e) \\ \vdots \\ C^{n_{\mathsf{en},p},e}(\boldsymbol{u}^e) \\ \boldsymbol{M}^{1,e}(\boldsymbol{u}^e) \\ \vdots \\ \boldsymbol{M}^{n_{\mathsf{en},v},e}(\boldsymbol{u}^e) \end{pmatrix} = \boldsymbol{0} \,. \tag{64}$$

Until now, the Jacobian $\mathbf{J}_{\boldsymbol{R}^e}$ of Equation (64) has always been implemented manually, but the aim of this thesis is, to also enable computational approaches, focusing on forward mode differentiation by operator overloading and forward finite differences. In order to compare the obtained results to a manual implementation, an analytical expression for the Jacobian has also been derived and implemented. The derivation can be found in Appendix A.

3.2. Architecture of CAMPIGA

CAMPIGA comes as a template library, which needs to be built from source before it can be linked to any user defined frontend. These frontends can be considered as specialized simulation tools tailored to a specific class of problems. Currently available are frontends for solving the 2D stationary incompressible NSE for a Newtonian fluid and for simulating structural problems governed by the linear elasticity equations in a Lagrangian frame of reference. These frontends use classes implemented in the CAMPIGA library and set up data structures required to solve the problem. Fig. 10 illustrates the architecture as well as the flow of information within CAMPIGA.

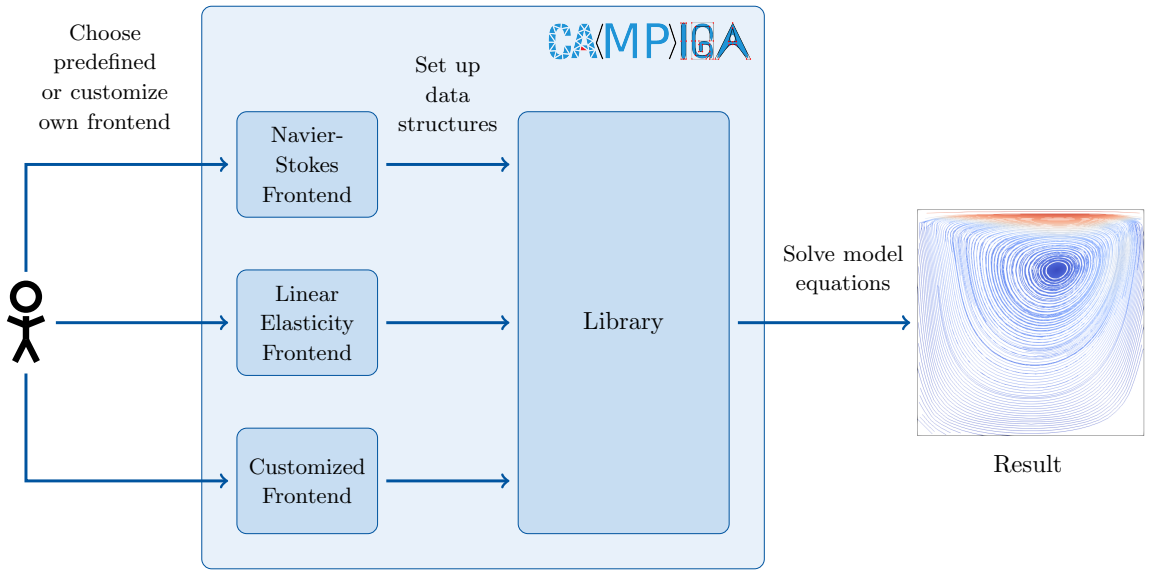


Figure 10: Architecture and flow of information within the CAMPIGA simulation software. The users typically only interact with one of the frontends, where they can either chose an already existing one or create a new one for their specific application. The frontend is then used to control the actual simulation.

The setup itself is split up into a compile-time and a run-time configuration. An overview of the most relevant compile-time and run-time settings can be found in Table 3.

The current implementation of CAMPIGA is realized with the aim to make frontends as flexible and readable as possible, so that users, who only want to run simulations, do not need to bother with modern aspects of the C++ programming language. However, this flexibility and clarity comes at the cost of some complexity in the underlying data structures, making the implementation of efficient AD or FD data types more challenging. The compile-time configuration for an example frontend is shown in Listing 5.

Compile-time Settings	Run-time Settings
spatial dimensionality	number of time steps, time step size
element types (triangles, quadrilaterals)	mesh
material laws	material parameters (density, viscosity)
model equations	solver tolerances
integration spaces, quadrature rules	maximum number of Newton iterations

Table 3: Overview of some common compile-time and run-time settings. The compile-time settings are always identical for a given frontend, while the run-time settings can be exchanged between different simulations with the same frontend.

Listing 5: Simplified compile-time settings for the Navier-Stokes frontend

```
// Data type specification
using TS = Scalar<>;
using TM = MultiDim<2, TS>;

// Settings for Navier-Stokes equations
using SpecificSettings = Settings<TM,
    Param<InstationaryMode::OFF>,    // stationary
    Param<Incompressible::ON>,       // incompressible
    Param<Stokes::OFF>               // with convection
>;

// Pick element type (Tria or Quad)
constexpr auto element_type = ElementType::Quad;

// Select quadrature order and rule
constexpr TM::Index quadr_order = 2;
using QuadRuleType = GaussQuadrature<element_type, quadr_order>;

// Choose physical material laws (constitutive equations)
using Laws = LawGroup<
    ViscosityNewton<TM>
>;

// Select equations and polynomial orders of ansatz functions
using ProblemType = DiffEquationGroup<TM,
    SpecificSettings, Laws, QuadRuleType,
    Momentum<LagrangeFE_2<element_type>>,
    Continuity<LagrangeFE_1<element_type>>
>;
```

The intrinsic data types, which are used for computations, are provided as aliases through wrapper classes. These wrapper classes are defined in the `type_spec` namespace. Encapsulating the information about the data types this way allows us to easily exchange them through customized types, e.g., ones that support an automatic determination of the Jacobian. Aliases to the used type specification structs for this example frontend are created in lines 2 and 3. The type specification structures are ordered hierarchically. The base class `Scalar` provides the data types for scalar variables. The `MultiDim` specification inherits from this scalar struct and extends the collection of data types by the corresponding vector and tensor classes. In this example and for the rest of this thesis, the number of spatial dimensions will be restricted to two dimensions.
The next five lines specify general settings concerning the class of problems, which should be solved with this frontend. The frontend presented in Listing 5 aims to solve the stationary, incompressible Navier-Stokes equations.
Line 13 specifies the type of mesh cells for the discretization of the computational domain. For two dimensional meshes, the user can choose either triangles or quadrilaterals. Afterwards the type of quadrature and its order are set. Since the type of the mesh cells as well as the quadrature are fully determined at compile-time, the number and the coordinates of the quadrature points are also known at compile-time. This offers the compiler more possibilities for optimization of the element-level assembly.
In lines 20 to 22, the constitutive law for the Cauchy stress tensor $\boldsymbol{\sigma}$ is provided. An important advantage of CAMPIGA's object-oriented code structure is that the material laws can be easily exchanged: Each material law is implemented in its own class. Each class fulfills a specific interface, e.g. all material laws formulated in the Eulerian frame of reference need to implement a method called `CauchyStress`, which takes a velocity gradient as input and returns the corresponding Cauchy stress tensor.
The core of the frontend implementation are the lines 25 to 29, where the problem itself is defined. The problem definition is stored in the `DiffEquationGroup` template. It requires all of the previously mentioned settings and additionally some structs, which provide information concerning the equations that are supposed to be solved. The information associated with a single equation is stored in an instance of type `DiffEquationDesc`. Each of them provides the equation for computing the contributions of a given element to the global residual vector and the global Jacobian.

For the extension of the existing code by an automatized matrix assembly, some requirements have been formulated:

- The implementation of the automatic matrix assembly should not require many changes of the existing architecture, especially it should work with the existing implementation of the `DiffEquationGroup` class. This class currently drives the element-level assembly of all equations involved in the problem.

- The usage of new methods for assembling the Jacobian should be integrated such that the user does not need to deal with the potentially complex code structure, enabling the use of automatized matrix assembly. Possible initializations of the CD capable data types are supposed to be hidden in the backend code to prevent simulations with incorrectly or incompletely initialized data structures.

To this end, the strategy for implementing an automatized matrix assembly by the means of forward mode algorithmic differentiation and forward finite differences can be summarized in the following steps:

1. Define suitable data types for computational differentiation, one each for AD (`AlgoDiffType`) and FD (`FiniteDiffType`). Both of them should be able to compute the value of an expression as well as its derivative.
2. Implement the corresponding type specification structs for computational differentiation. These structs should make the previously defined data types available in scalar and multi-dimensional data structures and allow for easy interchangeability.
3. Add a new class for equations (`CDEquation`), which do not provide an analytical expression for assembling their matrix manually, but determine them by some method of CD. This class will be designed as an adapter: It provides the same interface as the equations defined so far, but additionally accounts for the correct instantiation, initialization, and usage of the CD related data structures.
4. An additional class (`CDAssembly`) will be implemented as a wrapper for the frontend. It mimics the interface of the `DiffEquationDesc` structs but provides the specialized equation for computational differentiation. This new wrapper class can then be wrapped around any `DiffEquationDesc` in the frontend code to indicate that the contribution of a certain equation to the global Jacobian is assembled using one of the supported CD strategies.

The next Sections 3.3 and 3.4 will focus on the implementation of the data types for computational differentiation. After that, the required adaptions to the existing code will be explained in Section 3.5.

3.3. The `AlgoDiffType` Template

As mentioned before, the data type for algorithmic differentiation will use forward mode differentiation by operator overloading to compute the required Jacobian $\mathbf{J}_{\boldsymbol{F}}$. An algorithm for forward mode differentiation has already been proposed in Section 2.4,

Algorithm 1. Setting $\boldsymbol{x}^{(\mathsf{t})}$ to the i-th cartesian unit vector $\boldsymbol{e}_i \in \mathbb{R}^m$ will yield the partial derivative with respect to the input component x_i, this means

$$\boldsymbol{x}^{(\mathsf{t})} = \boldsymbol{e}_i \tag{65}$$

$$\Rightarrow \quad \boldsymbol{y}^{(\mathsf{t})} = \frac{\partial \boldsymbol{F}}{\partial x_i}\,. \tag{66}$$

In order to compute the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$ one needs to execute Algorithm 1 m times for $\boldsymbol{x}^{(\mathsf{t})} = \boldsymbol{e}_i$ with $i = 1, \ldots, m$. Clearly, each component $v_j^{(\mathsf{t})}$ of $\boldsymbol{x}^{(\mathsf{t})}$ has only once the value 1 (if $i = j$) and is zero otherwise. Thus, instead of evaluating Algorithm 1 m times, we can extend the algorithm to compute all derivatives simultaneously by exchanging each scalar $v_j^{(\mathsf{t})}$ by a vector. These vectors are initialized to the j-th cartesian unit vectors. That is, $\boldsymbol{x}^{(\mathsf{t})}$ becomes the m-dimensional identity matrix:

$$\mathbf{X}^{(\mathsf{t})} = (\boldsymbol{v}_1^{(\mathsf{t})}, \ldots, \boldsymbol{v}_m^{(\mathsf{t})}) = (\boldsymbol{e}_1, \ldots, \boldsymbol{e}_m) = \mathbf{Id} \in \mathbb{R}^{m \times m}\,. \tag{67}$$

The output $\boldsymbol{y}^{(\mathsf{t})}$ then also becomes a matrix, whose columns correspond to the partial derivatives of $\boldsymbol{F}$ with respect to each input x_j:

$$\mathbf{Y}^{(\mathsf{t})} = (\boldsymbol{v}_{m+p+1}^{(\mathsf{t})}, \ldots, \boldsymbol{v}_{m+p+n}^{(\mathsf{t})}) = \left(\frac{\partial \boldsymbol{F}}{\partial x_1}, \ldots, \frac{\partial \boldsymbol{F}}{\partial x_m}\right) = \mathbf{J}_{\boldsymbol{F}} \in \mathbb{R}^{n \times m}\,. \tag{68}$$

This leads to an adapted version of Algorithm 1, which will compute the full Jacobian. The new algorithm is shown in Algorithm 3.

Algorithm 3 Forward Mode Differentiation for computing the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$ of a function $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$

Input: $\boldsymbol{x}$
Result: $\boldsymbol{y}$, $\mathbf{J}_{\boldsymbol{F}}$
Start
$(v_1, \ldots, v_m) \leftarrow \boldsymbol{x}$ ▷ Copy states
$(\boldsymbol{v}_1^{(\mathsf{t})}, \ldots, \boldsymbol{v}_m^{(\mathsf{t})}) \leftarrow \mathbf{Id}$ ▷ Initialize derivatives
for $j = m+1, \ldots, m+p+n$ **do**
  $v_j \leftarrow \varphi_j(\{v_i\}_{i \prec j})$ ▷ Evaluate function
  $\boldsymbol{v}_j^{(\mathsf{t})} \leftarrow \sum_{i:\, i \prec j} \frac{\partial \varphi_j}{\partial v_i}\, \boldsymbol{v}_i^{(\mathsf{t})}$ ▷ Evaluate derivative
end for
$\boldsymbol{y} \leftarrow (v_{m+p+1}, \ldots, v_{m+p+n})$ ▷ Extract function value
$\mathbf{J}_{\boldsymbol{F}} \leftarrow (\boldsymbol{v}_{m+p+1}^{(\mathsf{t})}, \ldots, \boldsymbol{v}_{m+p+n}^{(\mathsf{t})})$ ▷ Extract Jacobian
End

This algorithm motivates the design of the `AlgoDiffType` class. In the implementation of the new data type, each component v_j will directly be associated with its derivative $\boldsymbol{v}_j^{(\mathsf{t})}$ by storing them together as member variables. All arithmetical operators and mathematical functions for this class need to be overloaded such, that for each elementary function φ_j the derivative $\boldsymbol{v}_j^{(\mathsf{t})}$ is computed together with the result v_j. The abbreviated class definition is shown in Listing 6.

Listing 6: Simplified class defintion of the `AlgoDiffType` class

```
template<typename T, unsigned int N_DERIVS>
class AlgoDiffType {
    // [...]
private:
    // [...]
    T v;                    // component value
    Vector<T,N_DERIVS> d;   // derivative vector
};
```

In order to be consistent with the classes implemented so far, the `AlgoDiffType` class is also templatized with respect to the data type `T`, that is used for the representation of numerical values. In practice, `T` will usually be `double`, but in case of memory-intensive computations, it can easily be exchanged for `float` in order to save memory. The second template parameter `N_DERIVS` refers to the number of active variables, which are total involved into the computation of $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$. When computing the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$, `N_DERIVS` is equal to the dimensionality of $\boldsymbol{x}$. The member variable `v` stores v_j, while the derivative $\boldsymbol{v}_j^{(t)}$ is stored in the vector `d`. Listing 7 shows the overloaded multiplication operation of two `AlgoDiffType` instances. In order to compute the derivative, the product rule of differential calculus is applied.

Listing 7: Simplified member implementation of the multiplication operator with an additional `AlgoDiffType` instance

```
//! Multiplication with AlgoDiffType
template<typename T, unsigned int N_DERIVS>
auto operator*(const AlgoDiffType<T,N_DERIVS>& other) {
    // temporary variable for storing the result
    AlgoDiffType<T,N_DERIVS> result;
    // compute the elemental function
    result.v = this->v * other.v;
    // compute its derivative by the product rule
    result.d = this->d * other.v + this->v * other.d;
    return result;
}
```

Let us consider again the example of the Rosenbrock function from Equation (12) to investigate the functionality of `AlgoDiffType`. A possible implementation of the corresponding SAC, which has been introduced in Equation (14), is shown in Listing 8. Through the use of templates, the implementation is kept general enough so that we can also apply it to the new `AlgoDiffType` class. The results for the example are listed in Table 4.

Listing 8: Sample implementation of the SAC representation of the Rosenbrock function. The implementation is templatized so that it can be computed using any data type providing the necessary arithmetical operators and mathematical functions.

```
template <typename T>
T rosenbrock(const Vector<T>& x) {
    T v1 = x[0];       // Assign the inputs to
    T v2 = x[1];       // temporary values.
    T v3 = pow(v1,2);
    T v4 = 1-v1;
    T v5 = v2-v3;
    T v6 = pow(v4,2);
    T v7 = pow(v5,2);
    T v8 = 100*v7;
    T v9 = v8+v6;
    T  y = v9;         // Store the output
    return y;
}
```

Instance	Member variable `v`		Member variable `d`	
	expression	value	expression	value
`x[0]`		-1		$[1,0]^T$
`x[1]`		1		$[0,1]^T$
`v1`		-1		$[1,0]^T$
`v2`		1		$[0,1]^T$
`v3`	`pow(v1.v,2)`	1	`2*pow(v1.v,1)*v1.d`	$[-2,0]^T$
`v4`	`1-v1.v`	2	`-v1.d`	$[-1,0]^T$
`v5`	`v2.v-v3.v`	0	`v2.d-v3.d`	$[2,1]^T$
`v6`	`pow(v4.v,2)`	4	`2*pow(v4.v,1)*v4.d`	$[-4,0]^T$
`v7`	`pow(v5.v,2)`	0	`2*pow(v5.v,1)*v5.d`	$[0,0]^T$
`v8`	`100*v7.v`	0	`100*v7.d`	$[0,0]^T$
`v9`	`v8.v+v6.v`	4	`v8.d+v6.d`	$[-4,0]^T$
`y`		4		$[-4,0]^T$

Table 4: Evaluation of the Rosenbrock function (Listing 8) for the instantiation `T=AlgoDiffType<double,2>`, in order to compute the derivative automatically. The member values of all involved (temporary) variables are shown.

The gradient is computed at position $\boldsymbol{x} = (-1, 1)^T$ and evaluates to $\boldsymbol{\nabla} f_{\text{Rb}}(\boldsymbol{x}) = (-4, 0)^T$. This result corresponds to the analytical result, which has been derived using adjoint mode differentiation in Table 2.

3.4. The `FiniteDiffType` Template

CAMPIGA offers an additional possibility for computing derivatives numerically. One sided forward differences (see Equation (21)) are applied for this purpose. Algorithm 4 shows, how Equation (21) can be applied to the SAC representation of a function.

Algorithm 4 Forward Difference approximation of a function's SAC representation

Input: $\boldsymbol{x}$, $\boldsymbol{x}^{(\mathsf{p})}$
Result: $\boldsymbol{y}$, $\tilde{\boldsymbol{y}}$
Start
$(v_1, \ldots, v_m) \leftarrow \boldsymbol{x}$ ▷ Copy states
$(v_1^{(\mathsf{p})}, \ldots, v_m^{(\mathsf{p})}) \leftarrow \boldsymbol{x}^{(\mathsf{p})}$ ▷ Copy perturbation
for $j = m+1, \ldots, m+p+n$ **do**
  $v_j \leftarrow \varphi_j(\{v_i\}_{i \prec j})$ ▷ Evaluate function
  $v_j^{(\mathsf{p})} \leftarrow \varphi_j(\{v_i^{(\mathsf{p})}\}_{i \prec j})$ ▷ Evaluate perturbation
end for
$\boldsymbol{y} \leftarrow (v_{m+p+1}, \ldots, v_{m+p+n})$ ▷ Extract function value
$\boldsymbol{y}^{(\mathsf{p})} \leftarrow (v_{m+p+1}^{(\mathsf{p})}, \ldots, v_{m+p+n}^{(\mathsf{p})})$ ▷ Extract perturbation
$\tilde{\boldsymbol{y}} \leftarrow \frac{\boldsymbol{y}^{(\mathsf{p})} - \boldsymbol{y}}{h}$ ▷ Compute Jacobian by forward differences
End

The vector $\boldsymbol{x}^{(\mathsf{p})}$ corresponds to the perturbed input. To compute the derivative of $\boldsymbol{F}$ with respect to input x_j, one needs to disturb the j-th component of $\boldsymbol{x}$ by the stepwidth h, this means:

$$\boldsymbol{x}^{(\mathsf{p})} = \boldsymbol{x} + h\, \boldsymbol{e}_j \tag{69}$$

$$\Rightarrow \quad \tilde{\boldsymbol{y}} = \frac{\boldsymbol{F}(\boldsymbol{x} + h\, \boldsymbol{e}_j) - \boldsymbol{F}(\boldsymbol{x})}{h} \approx \frac{\partial \boldsymbol{F}}{\partial x_j}\,. \tag{70}$$

If we are interested in the full Jacobian, we once again need to compute Algorithm 4 for all possible perturbations of $\boldsymbol{x}$, which means setting $\boldsymbol{x}^{(\mathsf{p})} = \boldsymbol{x} + h\, \boldsymbol{e}_j$ for $j = 1, \ldots, m$. However, we can introduce similar changes to the algorithm as in the previous chapter in order to compute all derivatives at once. Therefore, every $v_j^{(\mathsf{p})}$ is extended to a vector storing the perturbation $\boldsymbol{x} + h\, \boldsymbol{e}_j$. The variable $\boldsymbol{x}^{(\mathsf{p})}$ becomes a matrix:

$$\mathbf{X}^{(\mathsf{p})} = (\boldsymbol{x} + h\, \boldsymbol{e}_1, \ldots, \boldsymbol{x} + h\, \boldsymbol{e}_m) = \boldsymbol{x} \otimes \mathbf{1} + h\, \mathbf{Id}\,. \tag{71}$$

$\mathbf{1}$ denotes a vector, whose components are all equal to one. This change requires an adaption to the evaluation of the elemental function of the perturbed state. The elemental function φ_j needs to be evaluated component-wise for $\boldsymbol{v_i}^{(\mathsf{p})}$. We therefore introduce the notation

$$\boldsymbol{\Phi}_l(\{\boldsymbol{x}_l\}_{k \prec l}) = \begin{pmatrix} \varphi_l(\{x_{l,1}\}_{k \prec l}) \\ \vdots \\ \varphi_l(\{x_{l,m}\}_{k \prec l}) \end{pmatrix}\,, \tag{72}$$

where $x_{l,j}$ denotes the j-th component of the vector $\boldsymbol{x}_l$. In the extended algorithm, the result $\tilde{\boldsymbol{y}}$ also becomes a matrix

$$\tilde{\mathbf{Y}} = \left(\frac{\boldsymbol{v}_{m+p+1}^{(\mathsf{p})} - v_{m+p+1}\mathbf{1}}{h}, \ldots, \frac{\boldsymbol{v}_{m+p+n}^{(\mathsf{p})} - v_{m+p+n}\mathbf{1}}{h} \right) = \frac{1}{h} \left(\mathbf{Y}^{(\mathsf{p})} - \mathbf{1} \otimes \boldsymbol{y} \right) = \mathbf{J}_{\boldsymbol{F}} \,. \quad (73)$$

Here, we have $\mathbf{Y}^{(\mathsf{p})} = \left(\boldsymbol{v}_{m+p+1}^{(\mathsf{p})}, \ldots, \boldsymbol{v}_{m+p+n}^{(\mathsf{p})} \right)$. The complete algorithm for computing the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$ on the basis of a function's SAC representation is shown in Algorithm 5.

Algorithm 5 Finite Forward Differences for computing the full Jacobian $\mathbf{J}_{\boldsymbol{F}}$ of a function $\boldsymbol{y} = \boldsymbol{F}(\boldsymbol{x})$

Input: $\boldsymbol{x}$
Result: $\boldsymbol{y}$, $\mathbf{J}_{\boldsymbol{F}}$
Start
  $(v_1, \ldots, v_m) \leftarrow \boldsymbol{x}$ ▷ Copy states
  $(\boldsymbol{v}_1^{(\mathsf{p})}, \ldots, \boldsymbol{v}_m^{(\mathsf{p})}) \leftarrow \boldsymbol{x} \otimes \mathbf{1} + h\ \mathbf{Id}$ ▷ Initialize perturbations
  for $j = m+1, \ldots, m+p+n$ **do**
    $v_j \leftarrow \varphi_j(\{v_i\}_{i \prec j})$ ▷ Evaluate function
    $\boldsymbol{v}_j^{(\mathsf{p})} \leftarrow \boldsymbol{\Phi}_j(\{\boldsymbol{v}_i^{(\mathsf{p})}\}_{i \prec j})$ ▷ Evaluate perturbed function component-wise
  end for
  $\boldsymbol{y} \leftarrow (v_{m+p+1}, \ldots, v_{m+p+n})$ ▷ Extract function value
  $\mathbf{Y}^{(\mathsf{p})} \leftarrow \left(\boldsymbol{v}_{m+p+1}^{(\mathsf{p})}, \ldots, \boldsymbol{v}_{m+p+n}^{(\mathsf{p})} \right)$ ▷ Extract perturbations
  $\mathbf{J}_{\boldsymbol{F}} \leftarrow \frac{1}{h} \cdot \left(\mathbf{Y}^{(\mathsf{p})} - \mathbf{1} \otimes \boldsymbol{y} \right)$ ▷ Compute Jacobian by forward differences
End

To be consistent with the object-oriented implementation of forward mode differentiation, this functionality is also provided by a class. Each component v_j is therefore associated with its corresponding perturbation vector $\boldsymbol{v}_j^{(\mathsf{p})}$ and both are stored members of the new class `FiniteDiffType`. The finite difference approximation can then be computed on request. Similar to the `AlgoDiffType` class, `FiniteDiffType` also provides overloaded arithmetical operators and mathematical functions, which apply the elementary φ_j to the component and the perturbation vector in a component-wise fashion as well. A reduced class definition of `FiniteDiffType` is shown in Listing 9.

Listing 9: Simplified class definition of the `FiniteDiffType` class

```
template<typename T, unsigned int N_DERIVS>
class FiniteDiffType {
    // [...]
private:
    // [...]
    static constexpr double h = 1e-9;  // step width
    T v;                               // component value
    ValueGroup<T,N_DERIVS> p;          // perturbed vector
};
```

Member variable `v` stores the component v_j, the perturbed vector $\boldsymbol{v}_j^{(\mathrm{p})}$ is stored in member `p`, which is of type `ValueGroup`. This class implements a vector, whose arithmetic and mathematical operations are overloaded such that they are applied componentwise. The meaning of the template parameters is identical to class `AlgoDiffType`. Listing 7 shows how a multiplication operation between two `FiniteDiffType` instances can be overloaded.

Listing 10: Simplified member implementation of the multiplication operator with an additional `FiniteDiffType` instance

```
//! Multiplication with FiniteDiffType
template<typename T, unsigned int N_DERIVS>
auto operator*(const FiniteDiffType<T,N_DERIVS>& other) {
    // temporary variable for storing the result
    FiniteDiffType<T,N_DERIVS> result;
    // compute the elemental function
    result.v = this->v * other.v;
    // compute the perturbed elemental function
    result.p = this->p * other.p;
    return result;
}
```

As the implementation of a forward difference approximation of the Jacobian on a SAC level is not very intuitive, Table 5 presents an example, where the derivative of the multivariate Rosenbrock function is computed using its implementation from Listing 8. This time the function is instantiated with `T=FiniteDiffType<double,2>`.

Instance	Member variable `v`		Member variable `p`	
	expression	value	expression	value
`x[0]`		-1		$\begin{bmatrix} -0.9999 \\ -1 \end{bmatrix}$
`x[1]`		1		$\begin{bmatrix} 1 \\ 1.0001 \end{bmatrix}$
`v1`		-1		$\begin{bmatrix} -0.9999 \\ -1 \end{bmatrix}$
`v2`		1		$\begin{bmatrix} 1 \\ 1.0001 \end{bmatrix}$
`v3`	`pow(v1.v,2)`	1	`pow(v1.p,2)`	$\begin{bmatrix} 0.9998 \\ 1 \end{bmatrix}$
`v4`	`1-v1.v`	2	`1-v1.p`	$\begin{bmatrix} 1.9999 \\ 2 \end{bmatrix}$
`v5`	`v2.v-v3.v`	0	`v2.p-v3.p`	$\begin{bmatrix} 1.9999 \cdot 10^{-4} \\ 10^{-4} \end{bmatrix}$
`v6`	`pow(v4.v,2)`	4	`pow(v4.p,2)`	$\begin{bmatrix} 3.9996 \\ 4 \end{bmatrix}$
`v7`	`pow(v5.v,2)`	0	`pow(v5.p,2)`	$\begin{bmatrix} 3.9996 \cdot 10^{-8} \\ 10^{-8} \end{bmatrix}$
`v8`	`100*v7.v`	0	`100*v7.p`	$\begin{bmatrix} 3.9996 \cdot 10^{-6} \\ 10^{-6} \end{bmatrix}$
`v9`	`v8.v+v6.v`	4	`v8.p+v6.p`	$\begin{bmatrix} 3.999604 \\ 4.000001 \end{bmatrix}$
`y`		4		$\begin{bmatrix} 3.999604 \\ 4.000001 \end{bmatrix}$

Table 5: Evaluation of the Rosenbrock function (Listing 8) for the instantiation `T=FiniteDiffType<double,2>`, in order to compute the derivative automatically using a step width of $h = 10^{-4}$. The member values of all involved (temporary) variables are shown.

In contrast to the `AlgoDiffType` implementation, the perturbation vector `p` does not directly contain the forward difference approximation but only the perturbed function values. To compute the forward difference approximation of the derivative, we still need to subtract the function value, stored in the member variable `v`, and divide the difference by step width h. This yields for the example in Table 5

$$\nabla f_{\mathsf{Rb}}(\boldsymbol{x}) = \begin{pmatrix} \frac{3.999604-4}{1e^{-4}} \\ \frac{4.000001-4}{1e^{-4}} \end{pmatrix} = \begin{pmatrix} -3.96 \\ 0.01 \end{pmatrix}, \tag{74}$$

which is quite close to the analytical result $(-4, 0)^T$, given the relatively large step width.

3.5. Integration into the existing CAMPIGA framework

Now that we have some data structures that are capable of computing derivatives, we need to integrate them into the existing framework. The goal is that users should not need to bother with the selection and initialization of the correct data types if they want to assemble the Jacobian for their simulation by any means of CD. Instead, these steps should be managed by a separate class. Therefore, the `CDAssembly` template has been introduced together with two partial specializations, `ADAssembly` and `FDAssembly`, which determine, whether algorithmic differentiation or finite differences are used for the matrix assembly. By introducing this templatized struct, most of the frontend code presented in Listing 5 remains unchanged. In the definition of `ProblemType`, we can now wrap the new struct around the descriptions of the equations as shown in Listing 11:

Listing 11: Problem definition in the frontend after the introduction of CD capabilities

```
// Specify the assembly type for each equation individually
using ProblemType = DiffEquationGroup<TM,
    SpecificSettings, Laws, QuadRuleType,
    // Assemble Jacobian for momentum eq. by AD
    ADAssembly< Momentum<LagrangeFE_2<element_type>> >,
    // Jacobian for continuity eq. is provided manually
    Continuity<LagrangeFE_1<element_type>>
>;
```

If an equation should be assembled manually, one can still use the same syntax as in the introductory example. The flexibility that one can provide the assembly type for each equation separately comes in handy, as soon as more complex problems are solved: If one tries to solve problems, where an individual equation is very complex, e.g., due to the usage of a complicated material model, which makes the manual implementation of an analytical expression for the required Jacobian very tedious, it is possible to use

an automatic matrix determination only for this equation. Manual implementations can be used for the other equations to be more efficient.
The clarity of the frontend is retained through the use of a design pattern called *adapter*: The `CDAssembly` class inherits from the `DiffEquationDesc` struct, which contains the information associated with a distinct equation. Through the inheritance the `CDAssembly` class provides the same interface as the `DiffEquationDesc` struct. This allows it to be passed as a template argument to the `DiffEquationGroup`. However, the actual implementation of the interface can be adapted. The only requirement for the interface of the `CDAssembly` class is the provision of a type alias to an equation object. Here, the implementation of `CDAssembly` provides a specialized `CDEquation` instance, which contains the functionality for assembling the Jacobian automatically instead of evaluating the implementation of an analytical expression.

The `CDEquation` template is itself again implemented following the adapter pattern. It mimics the interface of any equation by providing an `Assemble` routine. This specialized `Assemble` method is now responsible for driving the automatic Jacobian determination. In order to distinguish between AD and FD, there exist two different specializations for this class, namely `ADEquation` and `FDEquation`. The program sequence of the specialized assembly strategy is similar for both cases and is visualized in Fig. 11.

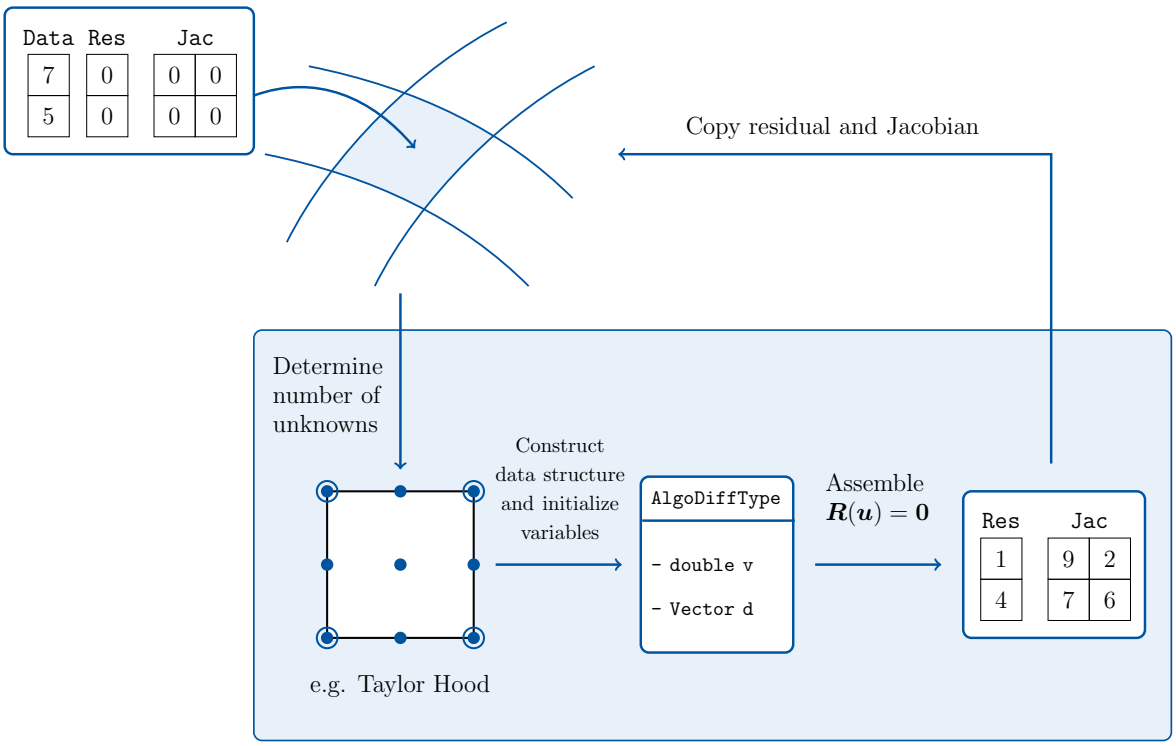


Figure 11: Program sequence for the case of an automatized matrix assembly.

As mentioned at the end of Section 3.1, the residual is assembled on an element-level basis. Thus, we also only need to calculate the element-level Jacobians. The first step now is to determine the number of unknowns on the element level. This number depends on the spatial dimensionality, the equations that are supposed to be solved, the chosen element type and the order of the ansatz functions. When solving the 2D NSE on Taylor-Hood elements (quadrilateral elements, quadratic ansatz functions for velocity, linear ansatz functions for pressure), the number of element-level unknowns amounts to 22 (see Fig. 12).

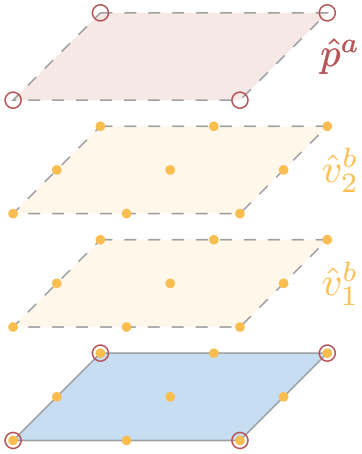


Figure 12: Degrees of freedom on a Taylor-Hood element for a two dimensional problem, where the velocity is discretized with quadratic ansatz functions, while a linear approximation is chosen for the pressure. This yields 18 unknown velocities and four unknown pressure values amounting to a total number of 22 unknowns.

This number is required to instantiate the correct CD type specification, which exchanges the default type for floating point numbers from `double` to `AlgoDiffType` or `FiniteDiffType` respectively. This type specification is then used to construct the data structure for storing all element-level quantities of interest, such as the current values for nodal unknowns, the ansatz functions or even the element-level residual. The residual and the nodal unknowns are now stored as variables of a CD type. The next step is to initialize these variables correctly, paying special attention to the correct initialization of the derivative as proposed in Algorithm 3 or Algorithm 5. After that, the assembly can be performed as before, with the only difference, that the Jacobian is now directly computed together with the residual. Once the assembly is finished, we only need to copy the residual as well as the Jacobian from the locally defined CD data structure into the one that has been passed from outside.

4. Computational Differentiation Results

This chapter presents two test cases, which have been simulated with the novel CAMPIGA framework. First, Section 4.1 presents the results for a two-dimensional flow between two parallel plates. The results for the velocity profile obtained with the different assembly strategies are compared to the analytical solution. Afterwards the performance of the discussed methods and their scaling for increasing problem sizes is investigated. Section 4.2 presents a cavity flow test case. The impact of different available assembly strategies on the convergence of the system is examined and measured by the number of Newton iterations needed to fulfill a certain threshold.

All test cases are governed by the two-dimensional, stationary, incompressible NSE. The following results are presented without units, as the test cases serve only for validation of the software and do not have any physical meaning.

4.1. Poiseuille Flow

The first test case simulates a steady, viscous flow between two parallel plates, also referred to as *Poiseuille flow* in the literature. The computational domain $\Omega = [0, 8] \times [-1, 1]$ is discretized by 30×30 Taylor-Hood elements (quadratic ansatz functions for velocity, linear ansatz functions for pressure). The grid and the boundary conditions are shown in Fig. 13.

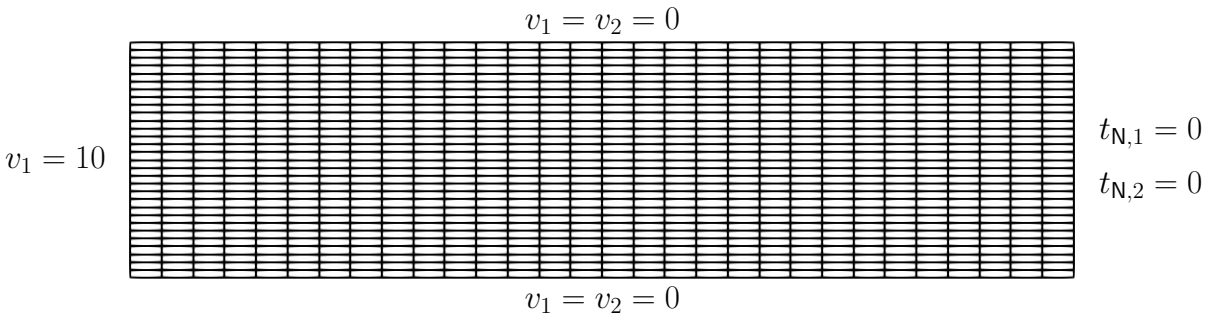
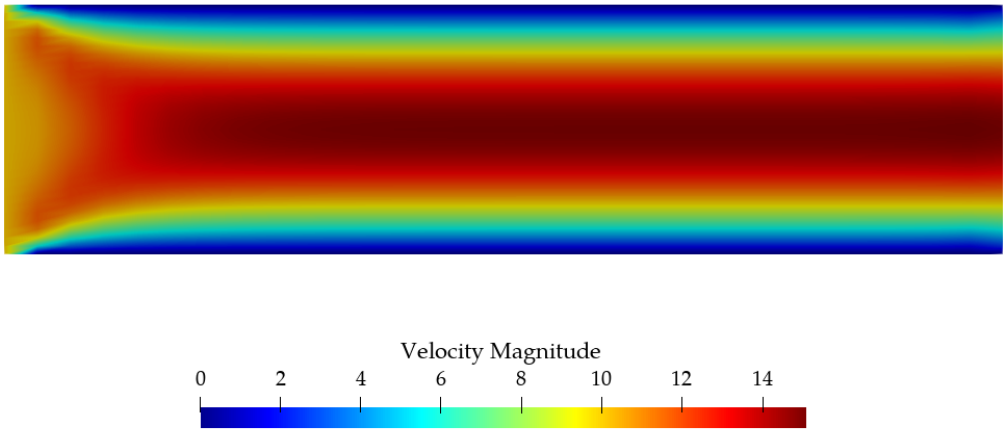


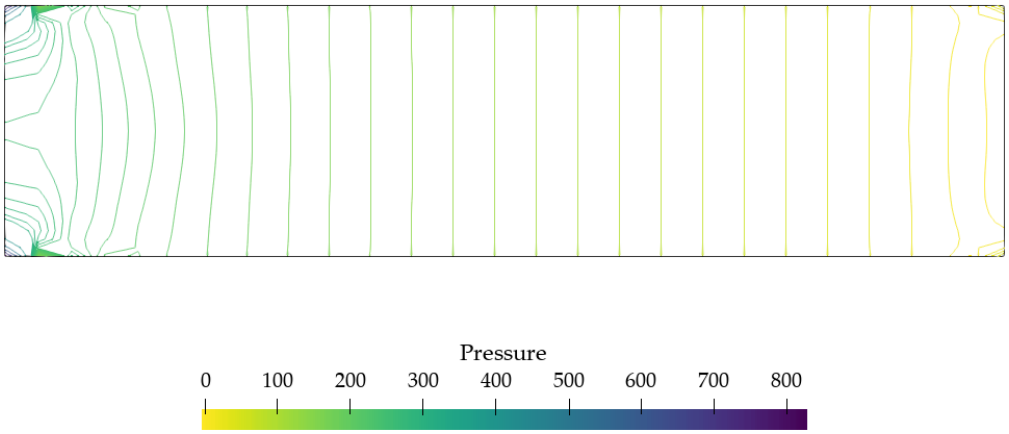
Figure 13: Grid and boundary conditions for the Poiseuille flow test case: No-slip conditions are prescribed on the top and bottom wall. A uniform velocity profile is chosen for the inflow (left). The natural Neumann conditions on the outflow are implicitly prescribed by the weak formulation.

To generate a flow, a uniform velocity profile is prescribed at the inflow on the left side. On the top and the bottom wall the velocity is set to zero. Due to this boundary conditions, we can expect that a parabolic velocity profile develops over the length of the inlet. At the outflow, natural Neumann boundary conditions are implicitly prescribed

by the implemented weak formulation: The contribution of the boundary integral, which would arise from the integration by parts, is neglected in Equation (47). This is mathematically equivalent to setting the traction $\boldsymbol{t}_{\mathrm{N}} = \mathbf{0}$ on the Neumann boundary. The simulation result for the velocity profile is shown in Fig. 14a. Fig. 14b shows the corresponding pressure field.



(a) Velocity field for the Poiseuille flow test case.



(b) Pressure field contours for the Poiseuille flow test case.

Figure 14: Results for the Poiseuille flow test case. The Jacobian has been assembled using a manual implementation.

The velocity field shows a short inlet section after which a parabolic velocity profile has developed. The pressure field exhibits some peaks at the upper and lower left corner of the computational domain. These can be related to the jump in the boundary conditions for the velocity at these corner points.

The NSE can be solved analytically for the case of a Poiseuille flow, see e.g. [37] for the derivation. The analytical solution for the horizontal velocity profile reads:

$$v_1(x_2) = \frac{1}{2\eta}\left(\frac{\partial p}{\partial x_1}\right)(x_2^2 - h^2)\,, \tag{75}$$

where h denotes the distance from the walls to the center-line between the plates. The pressure gradient will be approximated by the secant

$$\frac{\partial p}{\partial x_1} = \frac{\Delta p}{\Delta x}\,. \tag{76}$$

Δp and Δx will be measured by central differences between the nodes of two adjacent elements. Table 6 contains a list of all parameter values used for this test case.

Parameter	Value
ρ	1.00
η	1.00
Δp	-15.85
Δx	0.53
h	1.00

Table 6: Parameter values for the Poiseuille flow test case.

We can now compare the velocity profiles obtained by the discussed matrix assembly strategies to the analytical solution. They are shown in Fig. 15. The velocity profiles have been determined for a manual matrix implementation, as well as by AD and FD according to the description in Chapter 3. For the FD case, a step width of $h = 10^{-9}$ has been chosen. Clearly, the profiles generated by the three different assembly methods do not differ. There are only small deviations at the tip of the profile, where the peak velocity is reached. This difference can be related to the fact that the analytical solution holds true for a fully developed flow which is no longer influenced by the inlet. Although the velocity profile was measured downstream at $x = 6$, the numerical solution may still not be fully developed at that point, resulting in this slight difference. Fig. 16 compares the pressure along the flow channel. The analytical solution predicts a linear decay of the pressure. Once again, all three assembly methods show no significant differences. At the inlet, the numerical pressure results differ largely from the analytical pressure. However, the difference is limited to a very small interval at the inflow, where the flow is not fully developed. Further away from the inflow, the generated numerical results coincide with the analytical one.

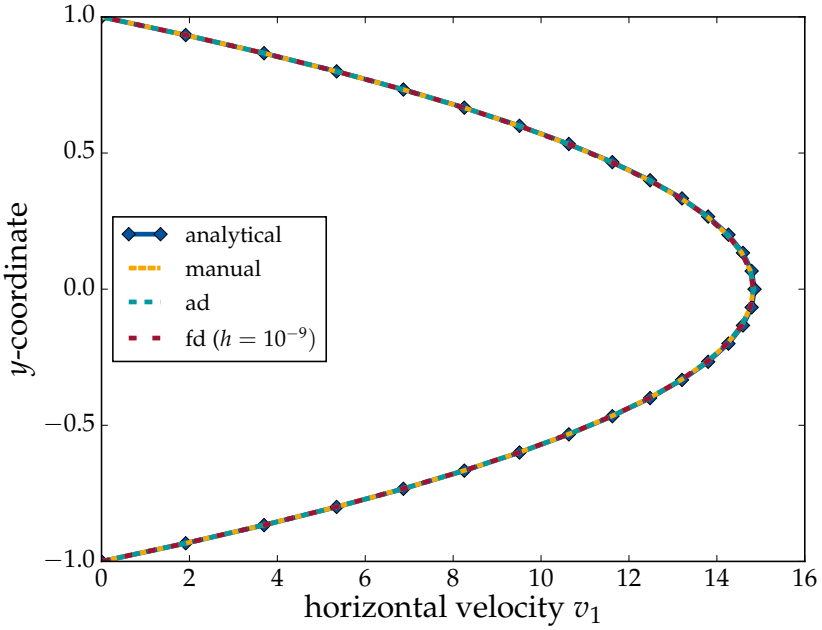


Figure 15: Comparison of the horizontal velocity profiles generated with the different assembly strategies and the analytical solution. The velocity profiles have been determined downstream at the location $x = 6$ in order to reduce the influence of the inlet path.

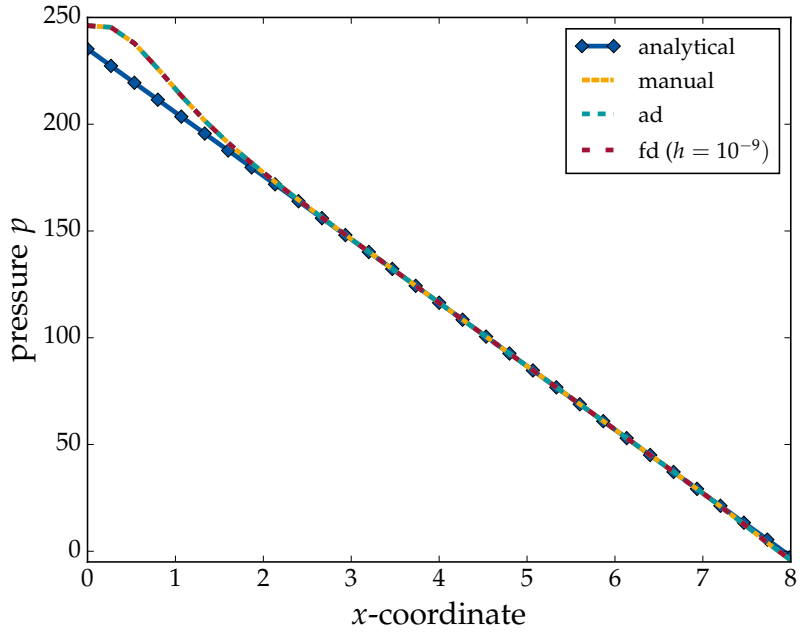


Figure 16: Comparison of the pressure drop in x-direction generated with the different assembly strategies and the analytical solution.

One of the key tasks that should be investigated in this thesis, was the performance of the presented assembly approaches. This has been done for the test case of the Poiseuille flow. The tests were executed on a login node of the RWTH compute cluster, which is equipped with an Intel Skylake Platinum 8160 CPU with 48 cores and a clock speed of 2.1 GHz[2].

The following aspects have been investigated: The test case has been simulated for two different mesh sizes, once using a mesh consisting of 100 elements, the second one using 400 elements. For the simulation, two different compiler optimization levels were used: One was compiled with the `-Og` option of `gcc` that is recommended by [1] as the highest level of optimization, which is still debugging-capable. The other one has been compiled with the `-O3` option offering the highest level of optimization available in `gcc`. All assembly methods have been applied successively. A list of the simulation cases is shown in Table 7:

Assembly method	Optimization level	Number of elements
Manual	`-Og`	100
Manual	`-Og`	400
Manual	`-O3`	100
Manual	`-O3`	400
AD	`-Og`	100
AD	`-Og`	400
AD	`-O3`	100
AD	`-O3`	400
FD	`-Og`	100
FD	`-Og`	400
FD	`-O3`	100
FD	`-O3`	400

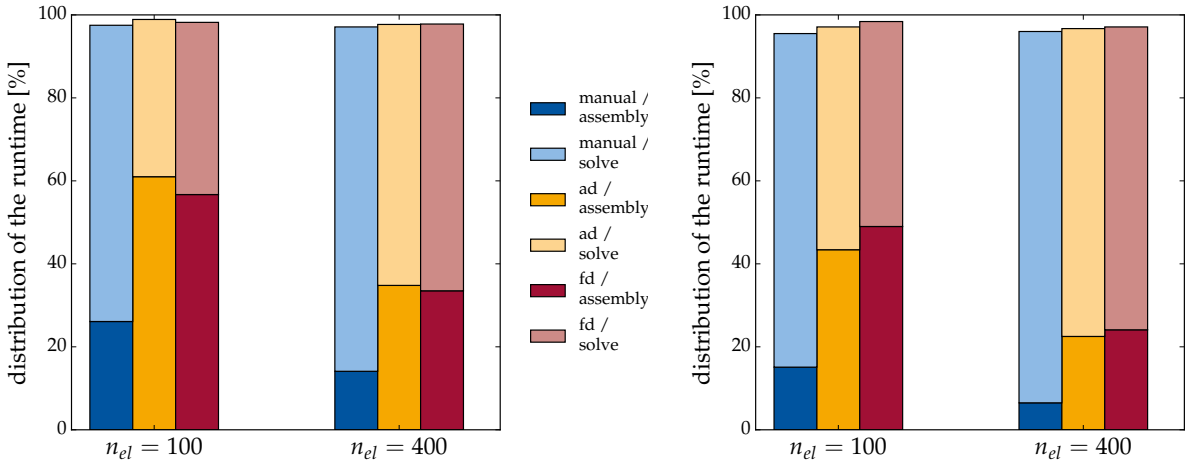
Table 7: Overview of simulation cases for the performance analysis.

The resulting simulations were profiled using the open-source tool `gprof`[3], which is able to measure the time a given executable spends in every function. Each of the simulation cases presented in Table 7 has been executed ten times in row and the results of each profile run have been averaged. From the `gprof` output, a list of functions was constructed that were mainly responsible for the measured run-time. Thereby, every function was considered, whose share of the whole run-time was greater than 1 %. Most of the functions in this list can be related to one of two main tasks: Either they contributed to setting up the linear system, that is determining the global residual

[2]see https://doc.itc.rwth-aachen.de/display/CC/Hardware+of+the+RWTH+Compute+Cluster, state 02.2020

[3]see https://sourceware.org/binutils/docs/gprof/, state 02.2020

$\boldsymbol{R}(\boldsymbol{u})$ of Equation (48) and its Jacobian $\mathbf{J}_{\boldsymbol{R}}$, or solving the resulting linear system given by Equation (53). Both tasks have been performed without any kind of parallelization. The measured run-times are reported in detail in Appendix B.
Fig. 17 shows, how the total run-time of the simulations is distributed among those two tasks. The darker colors represent the percentage for the assembly of the system, while the lighter colors symbolize the solution percentage.



(a) Without compiler optimization.

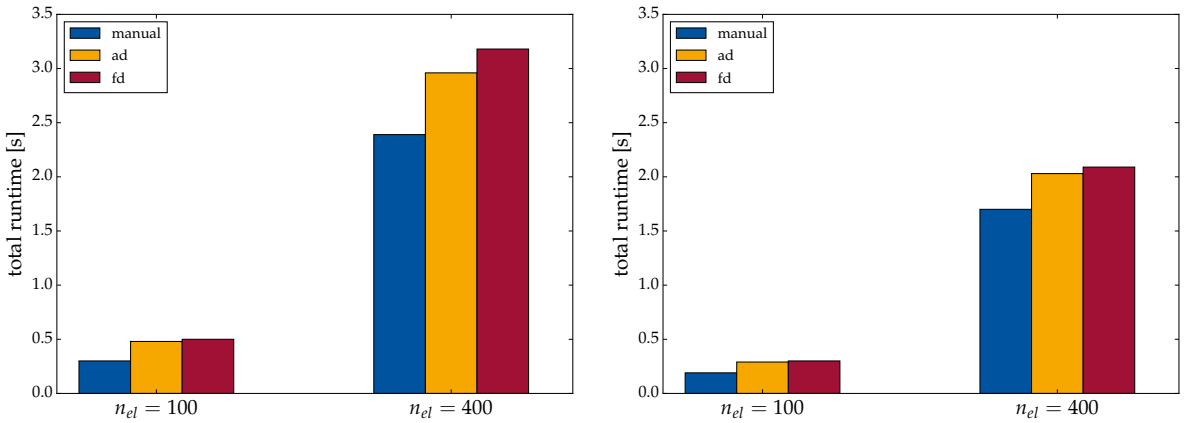
(b) With compiler optimization.

Figure 17: Assembly and solution percentage of the total run-time.

Fig. 17a shows the cases without compiler optimizations. One can see that for the smaller problem the automatic determination of the Jacobian (either with AD or FD) has the largest share of the total run-time. Clearly, as discussed in Section 2.1, an automatic determination of the Jacobian is less efficient than a manual implementation, which is also visible in Fig. 19. However, this effect is reduced when larger problems, e.g., $n_{\text{el}} = 400$, are considered, see Fig. 17a and Fig. 17b. This can be explained by different scalings: The assembly of the Jacobian scales in $\mathcal{O}(N^2)$, where N is the number of rows / columns. The solution of a linear system has in theory a complexity of $\mathcal{O}(N^3)$. However, most iterative solvers, such as the *BICGSTAB* method that is used here achieve a better performance in practice, especially, since the involved matrices are sparse. With compiler optimization, the portions of the run-time shift differently, see Fig. 17b. The percentage of the total run-time, which is due to the matrix assembly, is reduced. This can be explained by the different code architectures: The assembly part relies heavily on data structures whose sizes are known at compile-time. An example are the data structures for storing the element-level residuals and the element-level Jacobian: The problem dimensionality, the element shape and the ansatz functions are specified in the used frontend, which makes them available at compile-time. Thus the compiler has more flexibility regarding optimization and can unroll loops or replace function calls with parameters known at compile-time (such as e.g., the quadrature

points for the numerical integration) by the respective result. This is not as easily possible for the solution part of the code, which is currently provided by the external `Eigen`[4] library. The size of the matrix and the residual of the global Newton system depend on the actual mesh. Therefore, dynamic data structures have to be used to store them, as they can allocate the required amount of memory at run-time.

Fig. 18 depicts the total run-times for the simulations.



(a) Without compiler optimization.

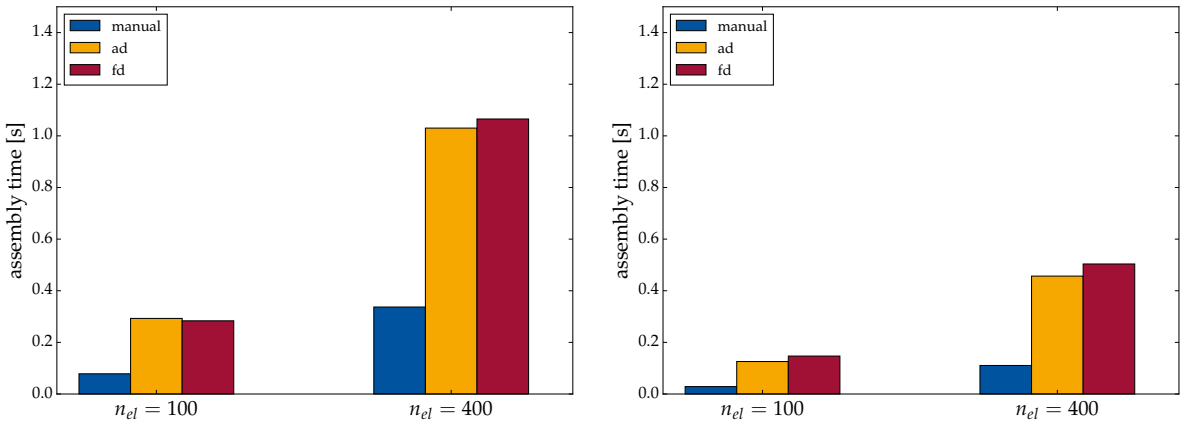
(b) With compiler optimization.

Figure 18: Total run-time for the different mesh sizes and assembly strategies.

Comparing Fig. 18a with Fig. 18b shows that the compiler optimization helps to reduce the overall run-time. The manual approach is always the fastest one, which coincides with our expectations. The CD-based approaches are slower, due to the overhead of computing the Jacobian. For the test case under consideration, the AD and FD approach perform similarly well. Thus one can recommend using the AD-based assembly, as this does not incorporate a truncation error, while being equally fast. Currently, with optimization, CD-based assembly methods are roughly $36\,\%$ faster for the smaller problem. However, for the larger number of elements, the benefit from compiler optimization is halved and only amounts to roughly $18\,\%$. Together with the diagram in Fig. 17, this strongly implies that for increasing problem sizes the influence of the different assembly strategies becomes less impactful compared to the time required to solve the linear system.

However, if one only compares the pure assembly times to each other, the performance gap between a manual and an automatized matrix implementation becomes visible. The results for the pure assembly times are shown in Fig. 19.

[4]see `http://eigen.tuxfamily.org/index.php?title=Main_Page`, state 02.2020

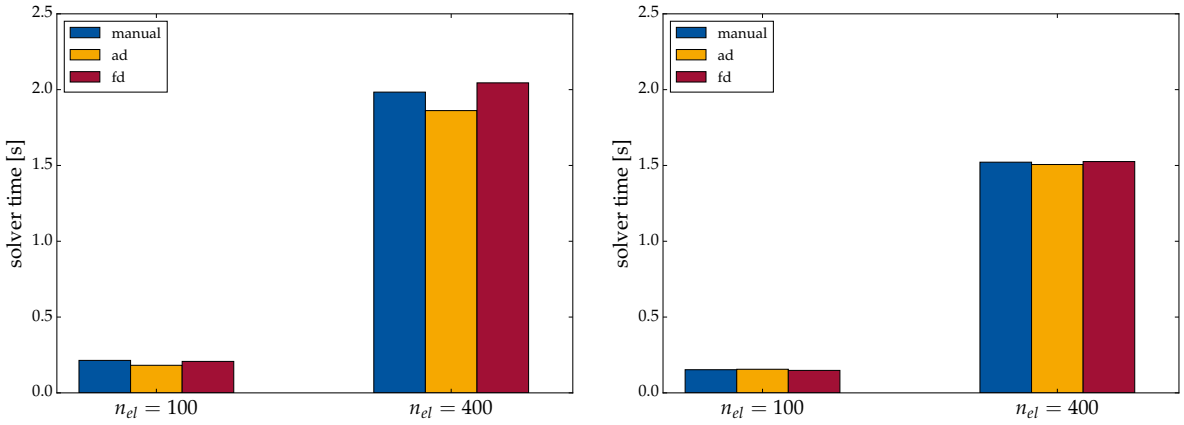


(a) Without compiler optimization.

(b) With compiler optimization.

Figure 19: Total time spent for the assembly of the Jacobians in each Newton iteration.

The times for the AD- and FD-based methods are very similar. The manual approach is three to four times faster depending on whether or not compiler optimization is used. This underlines the fact, that one should still provide a manual implementation whenever it is possible, especially for more complex problems, where the assembly times dominate the solution time. This statement is supported by Fig. 20, which shows a comparison of the total time required to solve the linear system.



(a) Without compiler optimization.

(b) With compiler optimization.

Figure 20: Total time spent for the solution of the linear system in each Newton iteration.

Independent of the compiler optimization, Fig. 20a and Fig. 20b show, that for all simulations the time needed to solve the linear systems within the Newton steps is quite similar. This leads to the conclusion, that for the examined test case the differences in the total run-times shown in Fig. 18 were completely determined by the choice of the assembly approach.

4.2. Lid-Driven Cavity

The next test case under consideration simulates a steady viscous cavity flow on the computational domain $\Omega = [0, 1]^2$. Similarly to the results presented in [13], the domain is discretized by 15×15 Taylor-Hood elements. The resulting mesh is shown in Fig. 21.

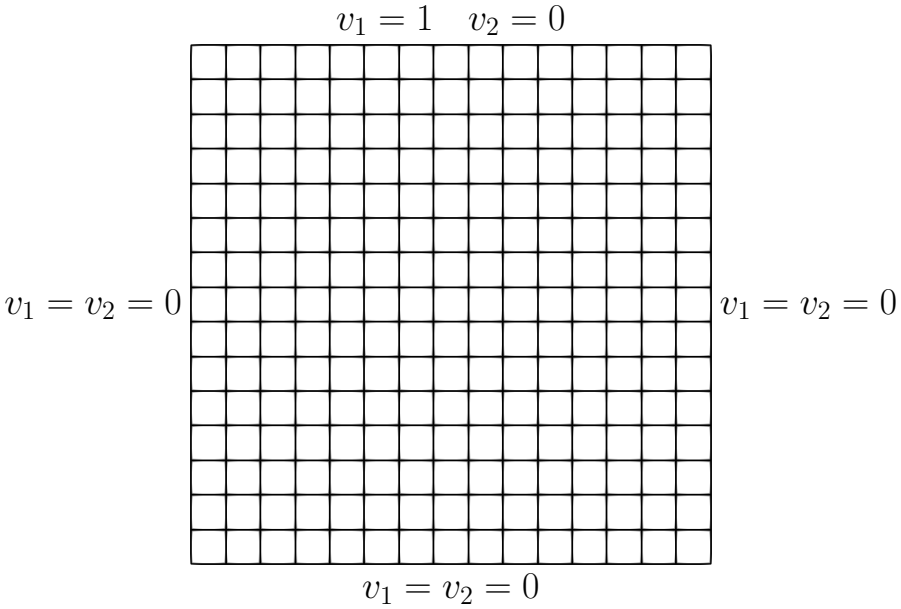


Figure 21: Grid and boundary conditions for the cavity flow test case: No-slip conditions are prescribed on all walls, except the top, where a constant velocity in positive x-direction is prescribed.

There are different ways, how the boundary condition for this test case can be set. For this thesis, a *leaky cavity* has been simulated, where on the whole top boundary (edges included) the non-zero horizontal velocity is prescribed.
We want to investigate a Stokes flow and a Navier-Stokes flow at a Reynolds number of $Re = 100$. A Stokes flow is a simplifications of the NSE presented in Equation (26), where the convective term $\boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v}$ is neglected. It can be derived from the NSE under the assumption of very small Reynolds numbers. The Reynolds number is defined as

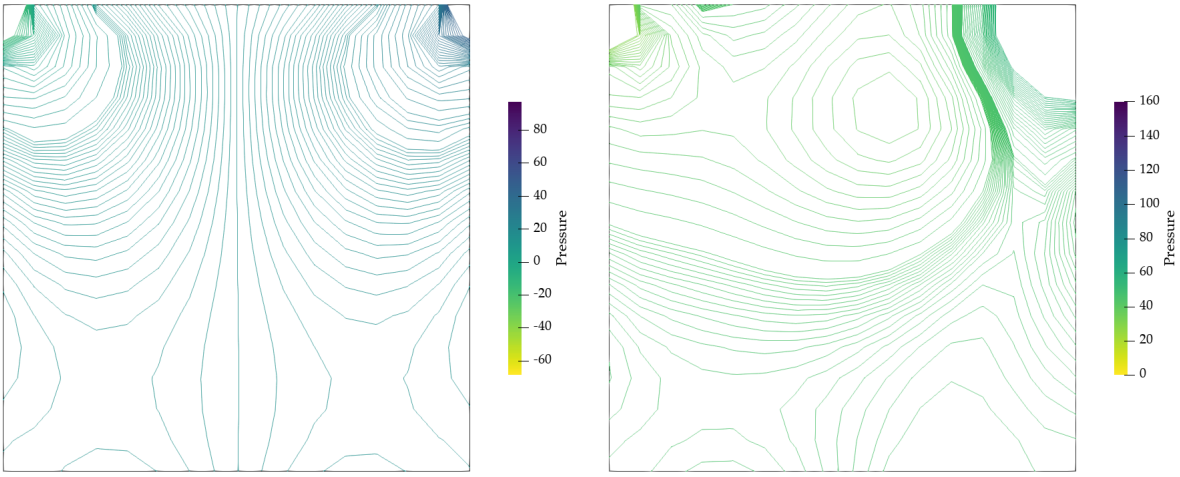
$$Re = \frac{\rho v_{\mathsf{ref}} l_{\mathsf{ref}}}{\eta}, \tag{77}$$

where for the cavity test case $v_{\mathsf{ref}} = 1$ (corresponding to the Dirichlet boundary condition on the top wall) and $l_{\mathsf{ref}} = 1$ (according to the width of the cavity) holds. The Reynolds number is a measure for the ratio of the inertia and friction forces.
The simulation results are evaluated for the pressure and the velocity fields. The material parameters used for the simulations are listed in Table 8.

Parameter	Value
ρ_{Stokes}	1.00
$\rho_{\text{Navier-Stokes}}$	100.00
η	1.00

Table 8: Parameter values for the lid-driven cavity test case.

For the presented selection of boundary conditions, the pressure is only determined up to a constant. Additionally, according to [13], oscillations in the pressure field can be expected if no stabilization is applied. However, the choice of Taylor-Hood elements for the discretization suffices to obtain reasonable pressure results. These are shown in Fig. 22.



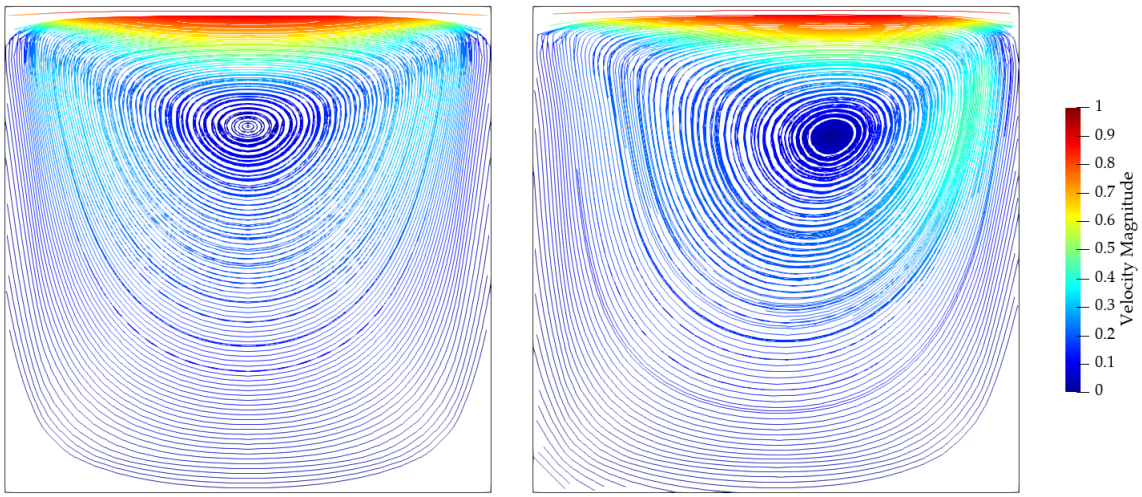
(a) Stokes.

(b) Navier-Stokes ($Re = 100$).

Figure 22: Comparison of the pressure field contours for the cavity flow test case.

In both cases, the pressure field contains two peaks in the upper left and right corner. These peaks can be related to the discontinuity in the boundary conditions at these points, where the moving lid passes the rigid walls. In the rest of the domain, both pressure fields are constant and do not exhibit any oscillations.

The streamlines for the Stokes and Navier-Stokes flow are shown in Fig. 23. For the case without convection, depicted in Fig. 23a, the streamlines are symmetric with respect to the vertical center line, the x-coordinate of the main vortex is $x = 0.5$. If we consider the influence of the convection term and increase the Reynolds number to 100, the location of the main vortex is moved to the left, see Fig. 23b. This corresponds to the expectations: A higher Reynolds number means that the friction forces are dominated by the inertial forces. Accordingly, the fluid behaves less viscously, which allows the main vortex to move to the right, following the motion prescribed on the top boundary.

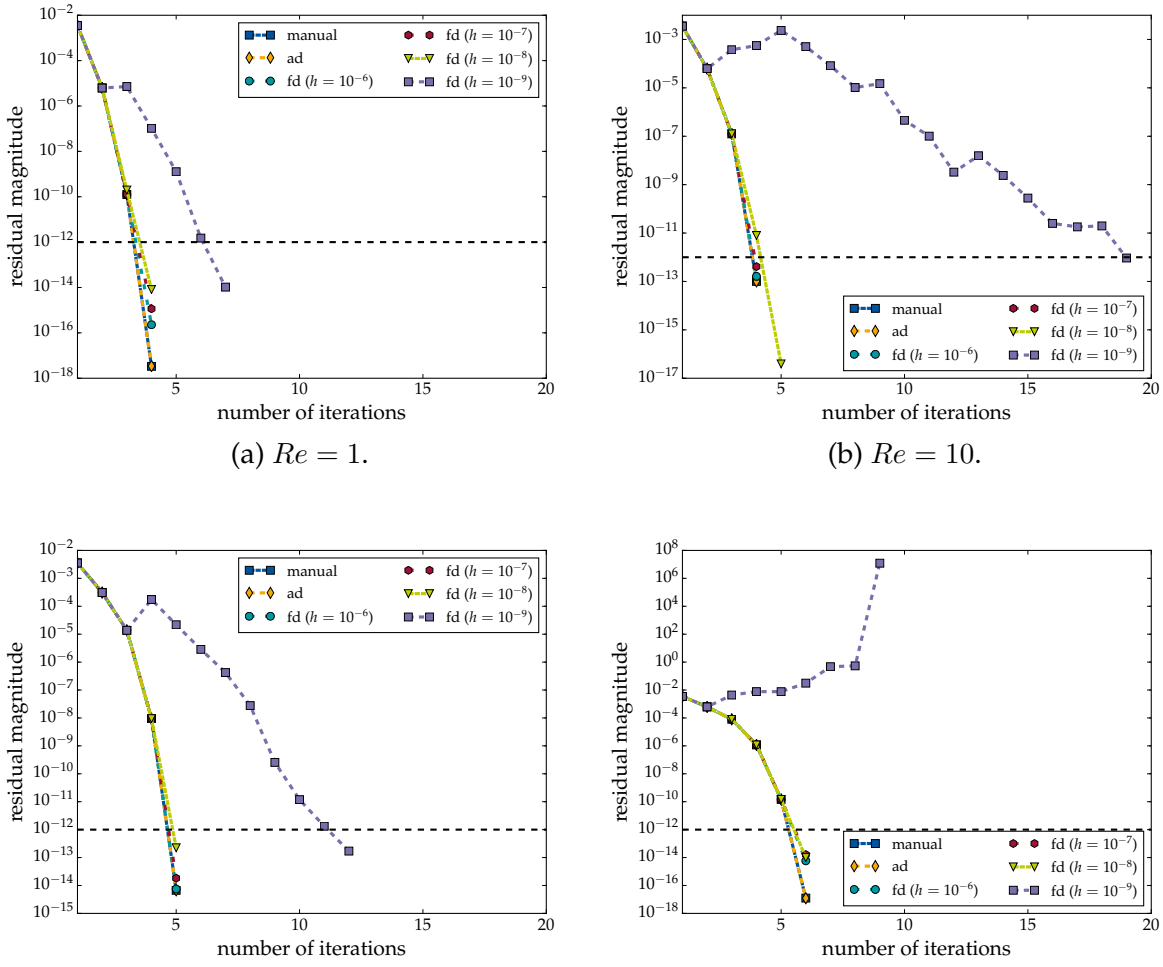


(a) Stokes. (b) Navier-Stokes ($Re = 100$).

Figure 23: Comparison of the streamlines and velocity magnitudes for the Cavity flow test case.

The cavity test case is used to examine the convergence properties of the discussed assembly strategies. We consider the Navier-Stokes equations and solve them for different Reynolds numbers by modifying the density. Each parameter setting is simulated for all implemented assembly types. For the FD approach, different step widths are chosen to investigate their influence on the convergence. The convergence is measured by the number of Newton iterations needed to push the norm of the residual below a given threshold.

The results are shown in Fig. 24. The dashed black line in each plot indicates the threshold below which the problem was considered to be converged. Clearly, all assembly types require the same amount of Newton iterations for this simple example. Only the FD assembly approach with a step width of $h = 10^{-9}$ always requires more iterations and does not converge for a Reynolds number of $Re = 100$, as depicted in Fig. 24d. This can be explained by numerical cancellation effects for very small step widths [35], which arise from a computer's inability to represent floating point numbers with arbitrary accuracy. The manual and the AD approach yield exactly the same convergence behavior, the obtained results for the residual are identical. This is to be expected, as both methods are exact with respect to machine precision.



(a) $Re = 1$.

(b) $Re = 10$.

(c) $Re = 50$.

(d) $Re = 100$.

Figure 24: Comparison of the convergence for the cavity test case in regimes of different Reynolds numbers. All assembly methods have been applied, the step width of the FD approach has been varied between 10^{-6} and 10^{-9}.

Part II.
Coupling Strategies

5. Overview of Coupling Strategies

When solving more complex engineering problems, one often finds that a specific problem is not only governed by the equation(s) for a single field of unknowns but instead is influenced by additional fields governed by different equations. In literature such problems are called *coupled problems*. One possible definition for this term is given by Felippa et al. [15] and has been adapted by Markert [27]:

> A *coupled problem* is one in which physically or computationally heterogeneous components interact dynamically. The interaction is multi-way in the sense that the solution has to be obtained by a simultaneous analysis of the coupled equations which model the problem.

Due to the mutual influence of the involved equations, the solution of such problems can become very challenging. There are many different examples of coupled problems from all fields of scientific research, including fluid-structure interaction, fluid-acoustic interaction, or acoustic-combustion interaction. Table 9 contains examples of research fields involving coupled problems and references to related papers, dealing with concrete problems of these fields.

Type of Interaction	Type of Coupling	Literature
Fluid-Concentration	Volume-Coupled	Zunino et al. [41]
Fluid-Structure	Surface-Coupled	Calo et al. [10], Nakata and Liu [28]
Fluid-Acoustic	Volume-Coupled	Schlottke-Lakemper et al. [36]
Acoustic-Combustion	Volume-Coupled	Lieuwen [25]

Table 9: Overview of research topics involving coupled problems.

There are different types of coupled problems. One typically distinguishes between volume-coupled and surface-coupled problems. While for the latter the domain of interaction is limited to the immediate interface of the subproblems, their volumetric counterparts describe physical phenomena, where different subproblems influence each other on the whole domain. For the rest of this thesis, we will restrict our considerations to volume coupled problems. There are different ways how coupled problems can be solved. In the following sections, related methods will be introduced and shortly explained.

5.1. Field Elimination

One possibility for solving coupled problems would be to eliminate one or more unknown fields by algebraic operations on the respective equations. To apply this technique, the model equations should have a sufficiently simple nature, which is typically not the case for models occurring in engineering applications. According to Markert [27], this approach originates from the times when such problems were solved manually instead of computationally and is no longer of use nowadays. Applying this method often leads to higher-order PDEs [15], which makes the solution more challenging from a numerical perspective.

5.2. Operator Splitting and Fractional Step Method

Operator splitting is another technique for solving coupled problems. The full complex problem is decomposed into multiple subproblems by splitting the differential operator into a series of simpler (e.g. one-dimensional) ones, which are then solved successively as depicted in Fig. 25. This way, each time step is divided into multiple (fractional) steps. There is a variety of time-stepping schemes, referred to as Fractional Step methods, which define different procedures for solving the series of differential operators within the discrete time interval. The kind of decomposition can directly account for field-specific properties [27]. For further information on this topic, we refer to the literature: Perot [32] presents a detailed analysis of the Fractional Step method, while Blasco et al. [7] discusses its application to the incompressible Navier-Stokes equations.

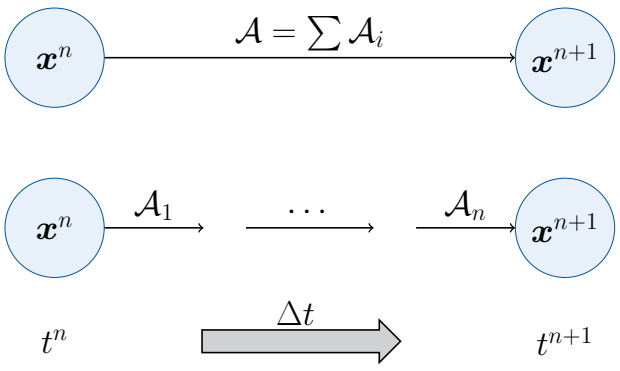


Figure 25: Instead of advancing the full differential operator over a time interval $[t^n, t^{n+1}]$, operator splitting decomposes the differential operator $\mathcal{A}$ into a series of simpler ones, which are then advanced over a fraction of the interval by Fractional Step methods.

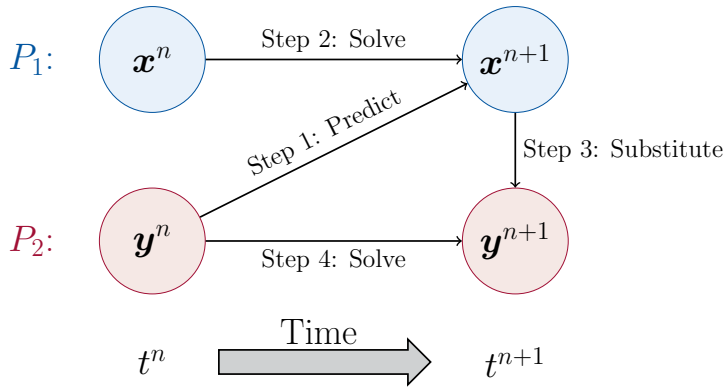
5.3. Monolithic Solution

A more general way of dealing with coupled problems is the monolithic solution approach. Here the whole system is treated as one quantity, and all equations are solved at the same time. To achieve numerical stability, usually, an implicit time-stepping scheme is applied. However, solving the resulting nonlinear system can become very challenging for engineering applications: On the one hand, the matrices become very large for three-dimensional problems. Additionally, the linear system, which is solved in a Newton step, can become ill-conditioned due to different orders of magnitude related to different fields. The development of preconditioners, which improve the condition of the resulting monolithic system, is an ongoing field of research. Some of these are presented by Verdugo and Wall [39]. The solution of a monolithic system is currently the choice in CAMPIGA.

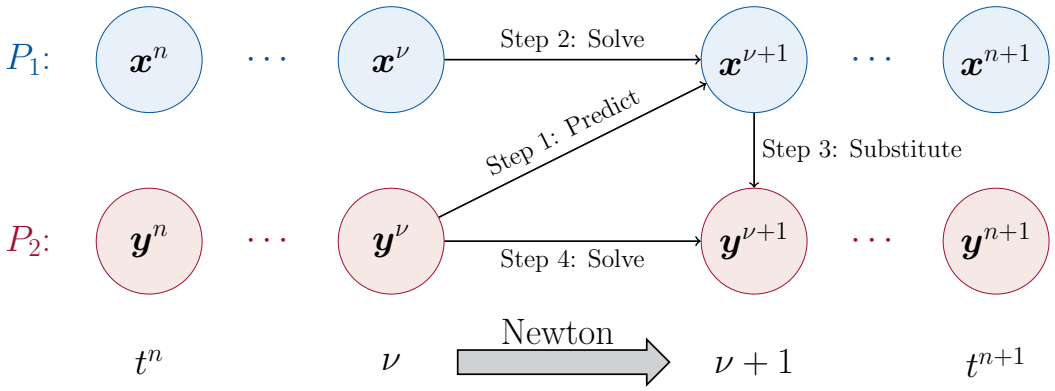
5.4. Partitioned Approach

The last method considered here for solving coupled problems is the so-called partitioned or staggered approach. The different models are treated as separate entities, whose temporal evolution can be chosen independently according to the field's properties. The interaction of the equations is realized in additional steps, which involve techniques such as prediction or substitution. This method is particularly valuable for problems where the physical phenomena take place on different time scales, such as in the field of fluid-structure interaction. The partitioned approach is very flexible and especially useful when combining different already existing simulation codes to access a new class of problems. Additionally, it results in smaller and better-conditioned subsystems compared to the monolithic approach. Felippa et al. [15] provide an overview of this broad topic. However, the implementation of staggered algorithms is no trivial task, and one needs to pay special attention to avoid a decrease in accuracy and stability [16].

One typically distinguishes between a weak or loose coupling and a strongly coupled approach. Both of them are schematically represented in Fig. 26. While the first one computes only one solution per field and time step (Fig. 26a), the latter employs additional sub-iterations to converge to the solution (Fig. 26b). Consequently, the weakly coupled approach requires less computational effort, at the cost of accuracy. Especially for complex problems, the weak coupling may face convergence issues, which makes the strong coupling the method of choice for the implementation in CAMPIGA.



(a) Weakly coupled approach.



(b) Strongly coupled approach.

Figure 26: Schematic representations of the different partitioned solution strategies for coupling two problems P_1 and P_2. In the weakly coupled case, the field data is only exchanged once between the two problems before the time step is advanced. In the strongly coupled case, the field data is exchanged repeatedly between the problems within a time interval. This data transfer happens within a Newton iteration.

6. Coupling Strategies in CAMPIGA

This chapter deals with the implementation of a strongly coupled solution approach within CAMPIGA. Section 6.1 introduces the equation used to examine the efficiency of the different available coupling strategies. After that, we focus on the algorithm for enabling a coupled solution approach within CAMPIGA and describe its integration into the existing software.

6.1. Instationary Model Equations

For testing the difference between a monolithic and partitioned solution approach, we slightly increase the complexity of the test problem presented in Section 3.1. We now consider the instationary NSE augmented by a heat conduction equation. The full test problem then reads:

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0 \qquad \text{in } \Omega \times [0, t_{\text{end}}] \tag{78}$$

$$\rho \left(\frac{\partial \boldsymbol{v}}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \right) - \boldsymbol{\nabla} \cdot \boldsymbol{\sigma} = \mathbf{0} \qquad \text{in } \Omega \times [0, t_{\text{end}}] \tag{79}$$

$$\rho c_p \left(\frac{\partial T}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} T \right) - \boldsymbol{\nabla} \cdot (\kappa \, \boldsymbol{\nabla} T) = 0 \qquad \text{in } \Omega \times [0, t_{\text{end}}] \tag{80}$$

$$\boldsymbol{v} = \boldsymbol{v}_{\text{D}} \qquad \text{on } \Gamma_{\text{D},v} \times [0, t_{\text{end}}] \tag{81}$$

$$\boldsymbol{n} \cdot \boldsymbol{\sigma} = \boldsymbol{t}_{\text{N}} \qquad \text{on } \Gamma_{\text{N},v} \times [0, t_{\text{end}}] \tag{82}$$

$$T = T_{\text{D}} \qquad \text{on } \Gamma_{\text{D},T} \times [0, t_{\text{end}}] \tag{83}$$

$$\boldsymbol{n} \cdot \boldsymbol{\nabla} T = q_{\text{N}} \qquad \text{on } \Gamma_{\text{N},T} \times [0, t_{\text{end}}] \tag{84}$$

$$\boldsymbol{v} = \boldsymbol{v}_0 \qquad \text{in } \Omega \times \{0\} \tag{85}$$

$$T = T_0 \qquad \text{in } \Omega \times \{0\} \, . \tag{86}$$

Here, T denotes the temperature, c_p the specific heat capacity and κ is the constant thermal conductivity. q_{N} is a surface-related heat flux. The equations presented above are only coupled in one direction: The instationary heat conduction equation (Equation (80)) depends on the solution of the momentum equation (Equation (79)) through the velocity $\boldsymbol{v}$, but the momentum equation does not depend on the solution of the heat equation. This could be changed by considering temperature dependent material properties, e.g. the viscosity. In order to reduce the complexity of this model, this feedback is not considered at this point.

In contrast to the equations presented in Chapter 3, the equations here are no longer stationary. This also has an impact on the discretization. In order to deal with the temporal derivative, we apply a semi-discrete scheme, following the method of lines: We first discretize the space using FEM as before. From this we obtain a large system of coupled ODEs in time which can be solved using a suitable time integration method.

Currently, an implicit Euler scheme is applied, which means, that every time derivative is discretized by backward finite differences in time. Given an ODE system of the form

$$\frac{\partial \boldsymbol{u}}{\partial t} = \boldsymbol{F}(\boldsymbol{u}, t)\,, \tag{87}$$

where $\boldsymbol{u}(t)$ is the vector of unknowns and $\boldsymbol{F}(\boldsymbol{u}, t)$ the right hand side, the implicit Euler method reads

$$\left.\frac{\partial \boldsymbol{u}}{\partial t}\right|_{t=t^{n+1}} = \boldsymbol{F}(\boldsymbol{u}^{n+1}, t^{n+1}) \tag{88}$$

$$\Rightarrow \frac{\boldsymbol{u}^{n+1} - \boldsymbol{u}^n}{\Delta t} = \boldsymbol{F}(\boldsymbol{u}^{n+1}, t^{n+1})\,. \tag{89}$$

$\boldsymbol{u}^n$ denotes the value of quantity $\boldsymbol{u}(t)$ at the time $t = t^n$. This backward Euler scheme is first order accurate in time, fully implicit and unconditionally stable, which means, that the time step width Δt can be chosen arbitrarily large [13].

The spatial discretizations for the continuity and momentum equation remain the same as presented in Chapter 3. The discretized weak formulation of the heat conduction equation can be derived analogously. Altogether, we obtain the following weak formulations for the transient case:

Find $(p^\mathsf{h}, \tilde{\boldsymbol{v}}^\mathsf{h}, \tilde{T}^\mathsf{h}) \in \mathcal{S}_p^\mathsf{h} \times \mathcal{S}_{\boldsymbol{v}}^\mathsf{h} \times \mathcal{S}_T^\mathsf{h}$ such that

$$\int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{v}^\mathsf{h})\, \phi_\mathsf{C}^A \,\mathrm{d}\Omega = 0 \tag{90}$$

$$\int_\Omega \rho \frac{\partial \boldsymbol{v}^\mathsf{h}}{\partial t}\, \phi_\mathsf{M}^B \,\mathrm{d}\Omega + \int_\Omega \rho\, (\boldsymbol{v}^\mathsf{h} \cdot \boldsymbol{\nabla} \boldsymbol{v}^\mathsf{h})\, \phi_\mathsf{M}^B \,\mathrm{d}\Omega + \int_\Omega \boldsymbol{\sigma}(p^\mathsf{h}, \boldsymbol{v}^\mathsf{h}) \cdot \boldsymbol{\nabla} \phi_\mathsf{M}^B \,\mathrm{d}\Omega = \boldsymbol{0} \tag{91}$$

$$\int_\Omega \rho c_p \frac{\partial T^\mathsf{h}}{\partial t}\, \phi_\mathsf{H}^C \,\mathrm{d}\Omega + \int_\Omega \rho c_p\, (\boldsymbol{v}^\mathsf{h} \cdot \boldsymbol{\nabla} T^\mathsf{h})\, \phi_\mathsf{H}^C \,\mathrm{d}\Omega + \int_\Omega \kappa\, (\boldsymbol{\nabla} T^\mathsf{h} \cdot \boldsymbol{\nabla} \phi_\mathsf{H}^C) \,\mathrm{d}\Omega = 0\,, \tag{92}$$

for all $(\phi_\mathsf{C}^A, \phi_\mathsf{M}^B, \phi_\mathsf{H}^C) \in \mathcal{V}_\mathsf{C}^\mathsf{h} \times \mathcal{V}_\mathsf{M}^\mathsf{h} \times \mathcal{V}_\mathsf{H}^\mathsf{h}$.

The function spaces $\mathcal{S}_T^\mathsf{h}$ and $\mathcal{V}_\mathsf{H}^\mathsf{h}$ are defined similar as in Section 3.1. The ansatz for the temperature reads:

$$T^\mathsf{h}(\boldsymbol{x}) = \underbrace{\sum_{c \in \eta_T \setminus \eta_{\mathsf{D},T}} \hat{T}^c\, \phi_T^c(\boldsymbol{x})}_{\tilde{T}^\mathsf{h}} + \underbrace{\sum_{c \in \eta_{\mathsf{D},T}} T_\mathsf{D}(\boldsymbol{x})\, \phi_T^c(\boldsymbol{x})}_{T_\mathsf{D}^\mathsf{h}}\,. \tag{93}$$

The weak formulation presented above, forms again a nonlinear equation system, which is also solved using Newton's method. For the monolithic case, the resulting Newton system becomes

$$\begin{pmatrix} \frac{\partial \boldsymbol{C}}{\partial \hat{\boldsymbol{p}}} & \frac{\partial \boldsymbol{C}}{\partial \hat{\boldsymbol{v}}} & \frac{\partial \boldsymbol{C}}{\partial \hat{\boldsymbol{T}}} \\ \frac{\partial \boldsymbol{M}}{\partial \hat{\boldsymbol{p}}} & \frac{\partial \boldsymbol{M}}{\partial \hat{\boldsymbol{v}}} & \frac{\partial \boldsymbol{M}}{\partial \hat{\boldsymbol{T}}} \\ \frac{\partial \boldsymbol{H}}{\partial \hat{\boldsymbol{p}}} & \frac{\partial \boldsymbol{H}}{\partial \hat{\boldsymbol{v}}} & \frac{\partial \boldsymbol{H}}{\partial \hat{\boldsymbol{T}}} \end{pmatrix} \cdot \begin{pmatrix} \Delta \hat{\boldsymbol{p}} \\ \Delta \hat{\boldsymbol{v}} \\ \Delta \hat{\boldsymbol{T}} \end{pmatrix} = - \begin{pmatrix} \boldsymbol{C}(\boldsymbol{u}) \\ \boldsymbol{M}(\boldsymbol{u}) \\ \boldsymbol{H}(\boldsymbol{u}) \end{pmatrix}\,, \tag{94}$$

where $\boldsymbol{u}$ is the vector which contains the nodal values of the unknowns

$$\boldsymbol{u} = (\hat{\boldsymbol{p}}, \hat{\boldsymbol{v}}, \hat{\boldsymbol{T}})^{\mathsf{T}} \tag{95}$$

$$= (\hat{p}^1, \ldots, \hat{p}^{n_{\mathsf{n},p}}, \hat{v}_1^1, \ldots, \hat{v}_{n_{\mathsf{sd}}}^1, \hat{v}_1^2, \ldots, \hat{v}_{n_{\mathsf{sd}}}^{n_{\mathsf{n},v}}, \hat{T}^1, \ldots, \hat{T}^{n_{\mathsf{n},T}})^{\mathsf{T}} \,. \tag{96}$$

The vectors $\boldsymbol{C}$, $\boldsymbol{M}$ and $\boldsymbol{H}$ represent all equations involved in the discrete problem

$$\boldsymbol{C} = (C^1, \ldots, C^{\mathsf{n},p})^{\mathsf{T}} \tag{97}$$

$$\boldsymbol{M} = (\boldsymbol{M}^1, \ldots, \boldsymbol{M}^{\mathsf{n},v})^{\mathsf{T}} \tag{98}$$

$$\boldsymbol{H} = (H^1, \ldots, H^{\mathsf{n},T})^{\mathsf{T}} \,, \tag{99}$$

where the components are the discretized weak formulations, multiplied by a distinct test function:

$$C^A(\boldsymbol{u}) = \int_\Omega (\boldsymbol{\nabla} \cdot \boldsymbol{v}^{\mathsf{h}})\, \phi_{\mathsf{C}}^A \, \mathrm{d}\Omega \tag{100}$$

$$\boldsymbol{M}^B(\boldsymbol{u}) = \int_\Omega \rho \, \frac{\partial \boldsymbol{v}^{\mathsf{h}}}{\partial t} \, \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega + \int_\Omega \rho \, (\boldsymbol{v}^{\mathsf{h}} \cdot \boldsymbol{\nabla} \boldsymbol{v}^{\mathsf{h}}) \, \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega + \int_\Omega \boldsymbol{\sigma}(p^{\mathsf{h}}, \boldsymbol{v}^{\mathsf{h}}) \cdot \boldsymbol{\nabla} \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega \tag{101}$$

$$H^C(\boldsymbol{u}) = \int_\Omega \rho c_p \, \frac{\partial T^{\mathsf{h}}}{\partial t} \, \phi_{\mathsf{H}}^C \, \mathrm{d}\Omega + \int_\Omega \rho c_p \, (\boldsymbol{v}^{\mathsf{h}} \cdot \boldsymbol{\nabla} T^{\mathsf{h}}) \, \phi_{\mathsf{H}}^C \, \mathrm{d}\Omega + \int_\Omega \kappa \, (\boldsymbol{\nabla} T^{\mathsf{h}} \cdot \boldsymbol{\nabla} \phi_{\mathsf{H}}^C) \, \mathrm{d}\Omega \,. \tag{102}$$

For the partitioned approach, we proceed as follows: Since for the incompressible case the continuity equation acts as a constraint for the momentum equation, these equations are still solved simultaneously. The heat conduction equation is solved in an additional step afterwards. For the coupled case, the corresponding Newton system reads

$$\begin{pmatrix} \frac{\partial \boldsymbol{C}}{\partial \hat{\boldsymbol{p}}} & \frac{\partial \boldsymbol{C}}{\partial \hat{\boldsymbol{v}}} & \boldsymbol{0} \\ \frac{\partial \boldsymbol{M}}{\partial \hat{\boldsymbol{p}}} & \frac{\partial \boldsymbol{M}}{\partial \hat{\boldsymbol{v}}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \frac{\partial \boldsymbol{H}}{\partial \hat{\boldsymbol{T}}} \end{pmatrix} \cdot \begin{pmatrix} \Delta \hat{\boldsymbol{p}} \\ \Delta \hat{\boldsymbol{v}} \\ \Delta \hat{\boldsymbol{T}} \end{pmatrix} = - \begin{pmatrix} \boldsymbol{C}(\boldsymbol{u}) \\ \boldsymbol{M}(\boldsymbol{u}) \\ \boldsymbol{H}(\boldsymbol{u}) \end{pmatrix} \,. \tag{103}$$

The next section is focused on the implementation of a strong coupling framework within CAMPIGA.

6.2. The `CoupledSystem` Template

Currently, a monolithic approach is the only option for solving multiple equations within CAMPIGA. One goal of this thesis is to extend the solution capability by a strong coupling framework, which allows to solve different equations or even whole problems iteratively. For this thesis, we restrict the number of coupled fields to two. In contrast to its weak counterpart, the strong coupling requires the exchange of data

between the coupled fields multiple times within the same time step. The data transfer will happen after each Newton iteration of a single field. Considering the system of the Navier-Stokes equations and the heat conduction problem, we can proceed as follows: Let $\boldsymbol{u}_1 = (\hat{\boldsymbol{p}}, \hat{\boldsymbol{v}})^\mathsf{T}$ denote the vector of unknowns of the first subsystem, while $\boldsymbol{u}_2 = \hat{\boldsymbol{T}}$ represents the nodal unknowns of the heat conduction problem. With $\boldsymbol{R}_1, \boldsymbol{R}_2$ and $\mathbf{J}_{\boldsymbol{R}_1}, \mathbf{J}_{\boldsymbol{R}_2}$ we refer to the residuals and Jacobians of the corresponding subproblems. Starting with the prediction $\boldsymbol{u}_{2,\text{pred}}$ for the temperature field, the velocity and pressure unknowns are computed from the corresponding linear system. For this specific problem, the temperature prediction would not be required, as the Navier-Stokes equations do not depend on the temperature. However, we include this step to make the coupling framework less restrictive. The results for pressure and velocity are then used to solve the heat conduction problem. The temperature values are updated and the next Newton iteration follows. This procedure can be summarized in the following algorithm:

Algorithm 6 Strong coupling of two interacting fields within a Newton iteration.

while $\|\boldsymbol{R}(\boldsymbol{u})\| > \varepsilon$ **do**
- $\boldsymbol{u}_{2,\text{pred}} \leftarrow \boldsymbol{u}_2^\nu$ ▷ Predict
- $\boldsymbol{u}^\nu \leftarrow (\boldsymbol{u}_1^\nu, \boldsymbol{u}_{2,\text{pred}})^\mathsf{T}$ ▷ Substitute
- $\mathbf{J}_{\boldsymbol{R}_1}(\boldsymbol{u}^\nu) \cdot \Delta\boldsymbol{u}_1^\nu = -\boldsymbol{R}_1(\boldsymbol{u}^\nu)$ ▷ Solve
- $\boldsymbol{u}_1^{\nu+1} \leftarrow \boldsymbol{u}_1^\nu + \Delta\boldsymbol{u}_1^\nu$ ▷ Update
- $\boldsymbol{u}^\nu \leftarrow (\boldsymbol{u}_1^{\nu+1}, \boldsymbol{u}_{2,\text{pred}})^\mathsf{T}$ ▷ Substitute
- $\mathbf{J}_{\boldsymbol{R}_2}(\boldsymbol{u}^\nu) \cdot \Delta\boldsymbol{u}_2^\nu = -\boldsymbol{R}_2(\boldsymbol{u}^\nu)$ ▷ Solve
- $\boldsymbol{u}_2^{\nu+1} \leftarrow \boldsymbol{u}_2^\nu + \Delta\boldsymbol{u}_2^\nu$ ▷ Update
- $\nu \leftarrow \nu + 1$

end while

For the prediction step of Algorithm 6 currently a zeroth order interpolation is chosen. However, one could also apply higher order predictors as mentioned in [16].

This algorithm has been implemented in CAMPIGA to allow the coupling of two arbitrary subproblems. Aiming for a non-invasive implementation of the novel coupling framework regarding the existing architecture and program procedures, an additional wrapper template `CoupledSystem` has been implemented. This template can be used in the frontend to define, which subproblems should be solved in a partitioned way. To this end, it is templatized with respect to the subproblems. A `CoupledSystem` instance provides the same interface as a single problem. This allows to solve monolithic systems as before, while simultaneously being able to solve coupled systems.

The data structures implemented so far, enable CAMPIGA to solve PDE systems using either the monolithic or the strongly coupled approach. The next chapter presents some test cases to validate the implementations and to compare the discussed coupling approaches with respect to their accuracy.

7. Coupling Strategies Results

The following chapter presents two test cases for verifying the newly implemented coupling framework of CAMPIGA. All examples are governed by the coupled system of the NSE and the heat conduction equation, presented in Section 6.1. The first test case is a modification of the already presented lid-driven cavity test case from Section 4.2. This test case has also been simulated with the XNS software, which is the state-of-the-art FEM solver at the institute and thus forms a good starting point for validation. The second test case examines a simple instationary Couette flow, which has been enhanced by additional temperature boundary conditions.

7.1. Lid-Driven Cavity

In the first coupling test case, we consider a square cavity on the domain $\Omega = [0, 1]^2$. For the discretization 30×30 elements are used. As before, velocity and pressure are discretized using the Taylor-Hood elements. For the temperature field, quadratic elements are selected. The mesh and the boundary conditions for this test case are shown in Fig. 27.

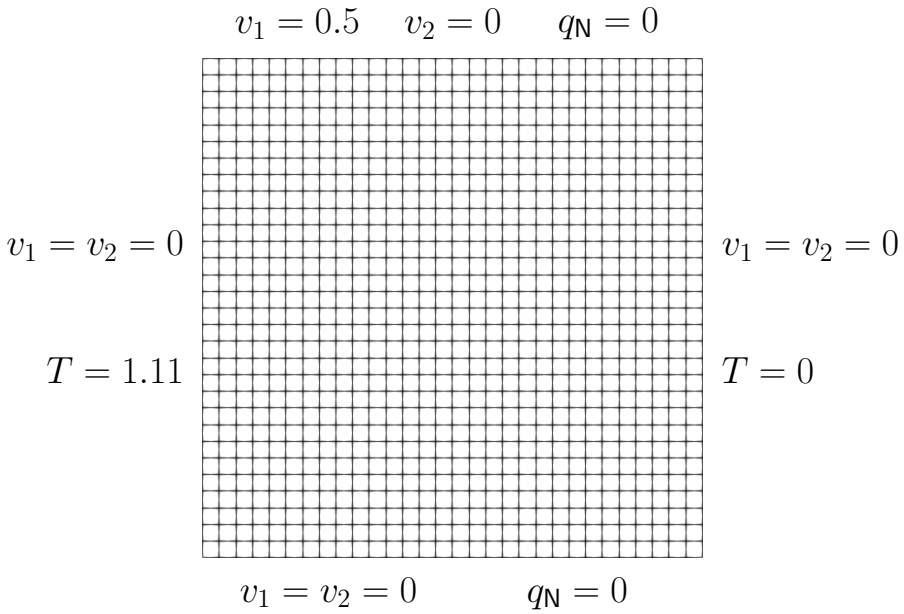


Figure 27: Grid and boundary conditions for the coupled cavity flow test case: No-slip conditions are prescribed on all walls, except the top, where a constant velocity in positive x-direction is prescribed. On the left and the right wall constant temperatures are prescribed. The top and bottom wall are adiabatic.

The mesh used for the simulation in XNS has the same number of elements, but they are distributed differently as shown in Fig. 28. The mesh is refined at the walls of the cavity, which allows resolving both the velocity and the temperature boundary layer more accurately.

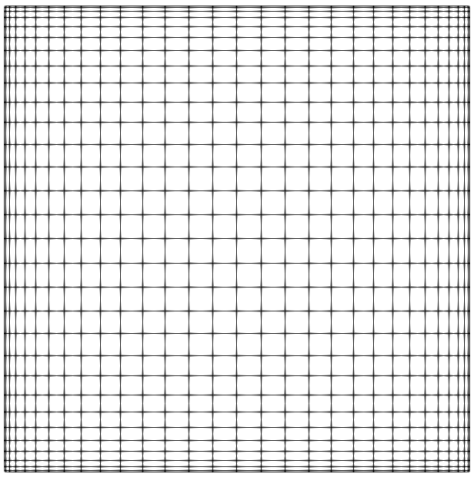


Figure 28: Mesh used to obtain the reference solution with XNS.

The problem has been simulated with XNS for the stationary case, so we neglect the temporal derivatives in Equation (79) and Equation (80). The values for the material parameters are listed in Table 10.

Parameter	Value
ρ	10.10
η	0.05
c_p	1.43
κ	1.40

Table 10: Parameter values for the coupled lid-driven cavity test case.

In order to verify the implementation of both the monolithic and the coupled approach, we compare the results for the velocity and the results for the temperature obtained with CAMPIGA to the reference solution of XNS. The reference solution for the velocity field is presented in Fig. 29.

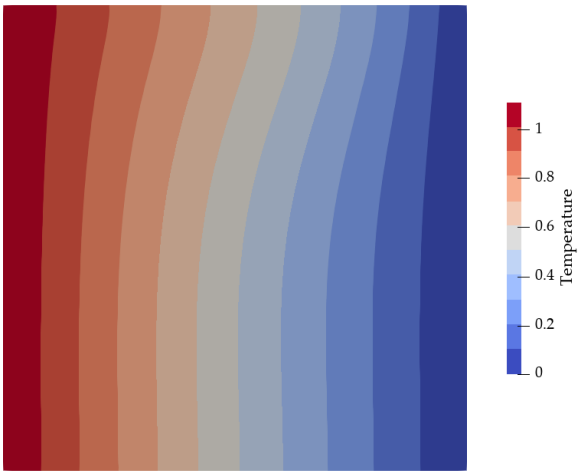


Figure 29: Reference temperature field obtained from a simulation with XNS.

The same temperature field is obtained by the monolithic and the staggered approach. For a better comparison, we plot the temperature distribution along the horizontal line $y = 0.9$, for all simulations. The resulting temperature profile is depicted in Fig. 30.

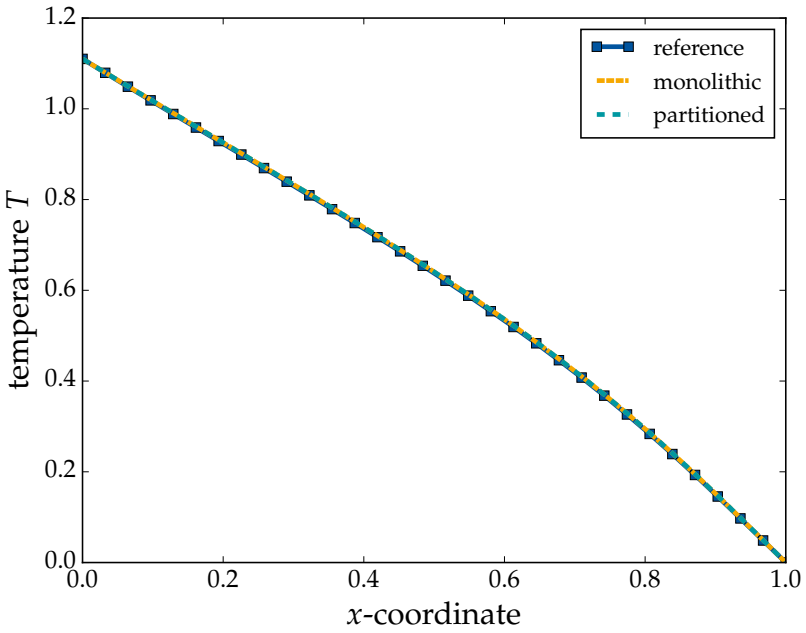


Figure 30: Temperature profiles along the horizontal line $y = 0.9$ obtained with XNS and CAMPIGA.

The results are identical independent of the chosen coupling approach. Since the left boundary is heated with a constant temperature $T = 1.11$, the fluid temperature is also larger near the left wall. At the right wall, the temperature decays faster than before.

This can be related to the motion of the fluid: The prescribed velocity of the lid results in the development of a vortex. Near the right wall, the horizontal velocity is much smaller than the vertical velocity, meaning that less heat from the left side is convected to the right boundary. This results in a lower temperature due to the cooling of the right wall.

Additionally, the velocity fields obtained with the different coupling methods have been compared. The result is shown in Fig. 31.

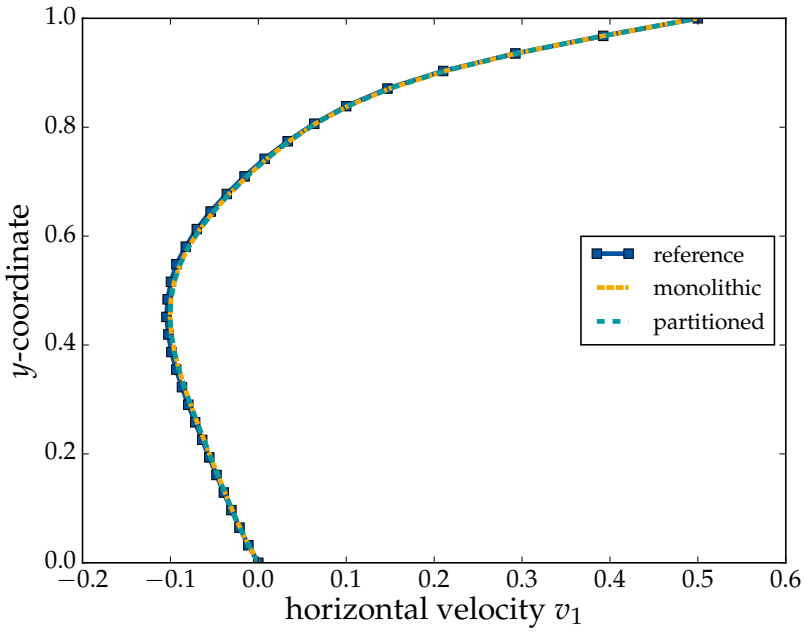


Figure 31: Horizontal velocity profiles along the vertical line $x = 0.5$ obtained with XNS and CAMPIGA.

There is no significant deviation between the discussed approaches. The results for the horizontal velocity field look qualitatively similar to velocity profiles from [13], where this test case has been investigated for different Reynolds numbers. This suggests, that the implemented algorithms produce correct results.

7.2. Couette Flow

With the second test case, we now examine the influence of the chosen coupling strategy on the amount of Newton iterations necessary to obtain a converged solution. For that, we consider the setting of a Couette flow, but apply additional temperature boundary conditions, as depicted in Fig. 32.

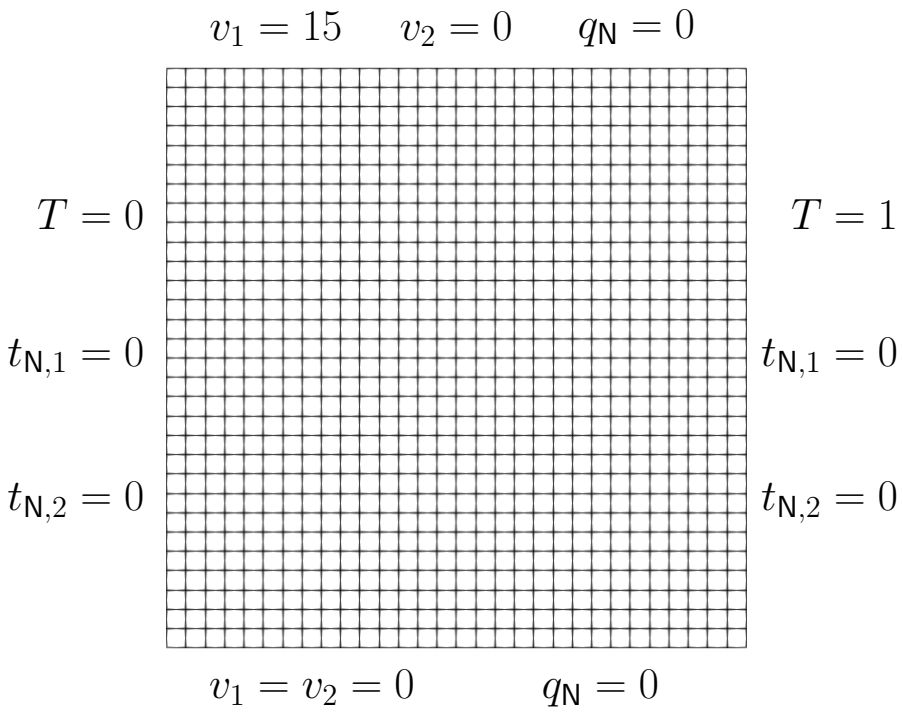


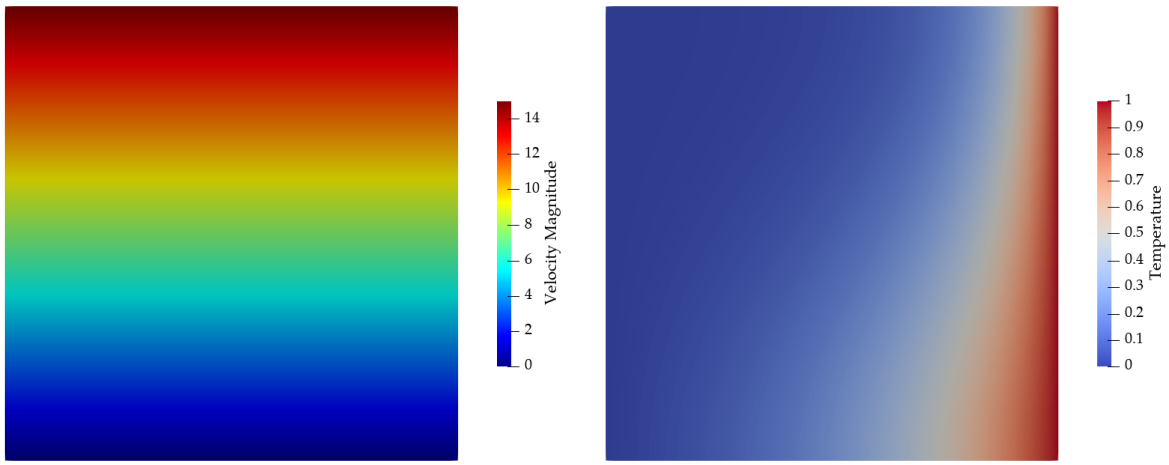
Figure 32: Grid and boundary conditions for the coupled Couette flow test case: No-slip conditions are prescribed on the bottom. The top wall is moved with a constant velocity in positive x-direction. The outflow is heated with a constant temperature, so that the back propagation of the temperature against the flow direction can be examined. The top and the bottom wall are adiabatic.

This time, the transient model equations are solved. For the temporal discretization, we choose a time step width of $\Delta t = 0.1$ and advance the equations for a total amount of $n_{\mathsf{ts}} = 10$ time steps. The material parameters selected for the simulation are listed in Table 11.

Parameter	Value
ρ	1.0
η	1.0
c_p	1.0
κ	1.5

Table 11: Parameter values for the coupled Couette flow test case.

The analytical solution (see [37] for a derivation) predicts a linear velocity profile for this test case. The simulation results for the velocity and temperature field obtained with the monolithic approach are shown in Fig. 33. The simulation using the coupled approach yields similar results.



(a) Velocity field for the Couette flow test case.

(b) Temperature field for the Couette flow test case.

Figure 33: Results for the coupled Couette flow test case at $t_{\text{end}} = 1$.

The velocity profile is perfectly linear for both coupling strategies, see also Fig. 34, where the horizontal velocity has been plotted over the height of the flow channel. The simulation results coincide with the analytical solution.

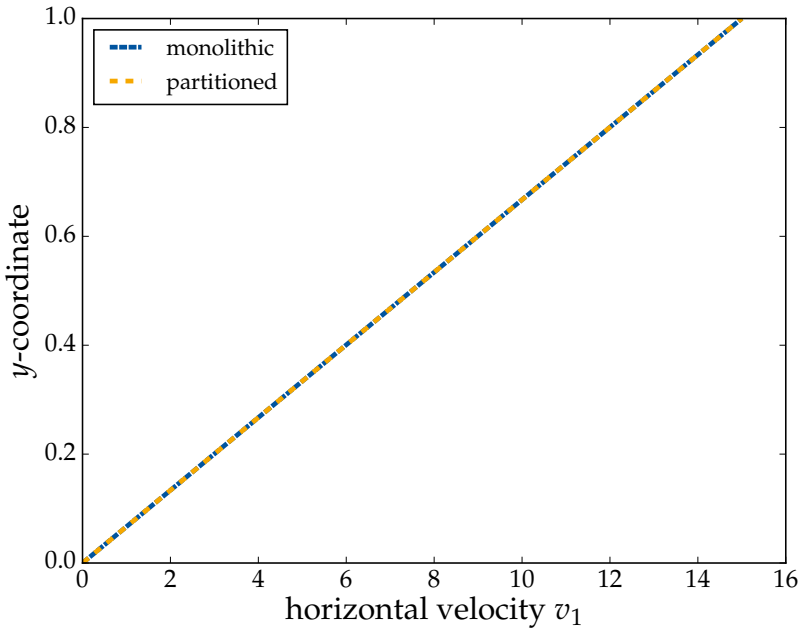


Figure 34: Comparison of the horizontal velocity profiles obtained with the different coupling strategies. The velocity profiles have been determined along a vertical line at $x = 0.5$.

The temperature profile depicted in Fig. 33b can be explained as follows: The top of the flow channel is moved with a constant velocity, while a no-slip condition is prescribed at the bottom. Consequently, the fluid moves faster near the top wall than near the bottom wall. Since the outflow is heated in this test case, the heat has to propagate upstream, against the direction of flow. Accordingly, heat can easier propagate on the bottom side due to the lower velocity. Therefore, the temperature is higher on the bottom side than on the top side of the fluid. Fig. 35 shows the temperature profile along a vertical line, located at $x = 0.9$.

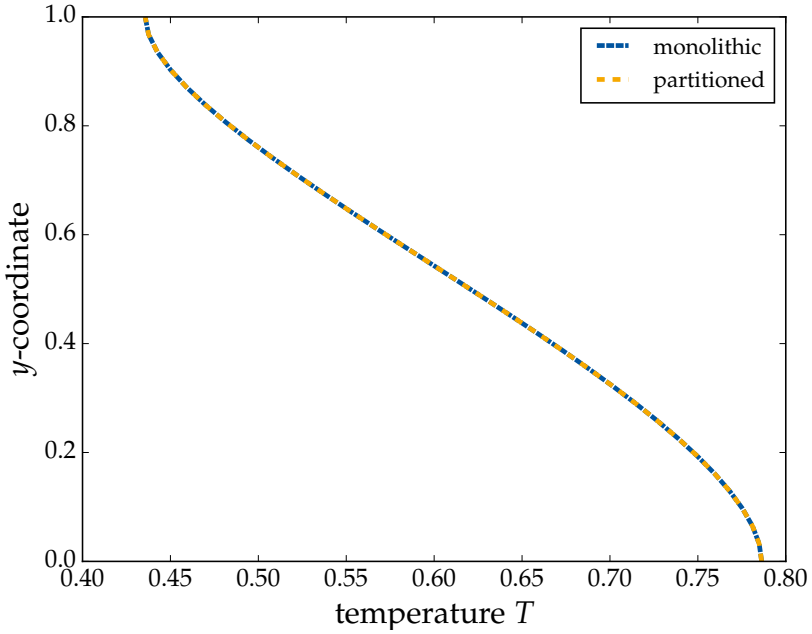


Figure 35: Comparison of the temperature profiles obtained with the different coupling strategies. The temperature profiles have been determined along a vertical line near the outflow at $x = 0.9$.

Once again, both coupling strategies produce completely identical results.

One aspect, which is of particular interest, when dealing with coupling strategies, is their impact on the number of Newton iterations needed to obtain a converged solution. This has also been examined for this test case and the results are shown in Fig. 36. Due to the simplicity of the test case, which has no significant nonlinearities, the monolithic approach mostly converges after two iterations. However, for this special problem, the partitioned approach requires even fewer Newton steps. This can be related to the uni-directional dependence: The heat conduction equation depends on the solution of the NSE but not vice-versa. Solving the heat conduction problem in a separate step has the additional advantage that this problem becomes linear, once the velocity is no longer considered an unknown. This explains, why the solution of the heat conduction requires mostly only one single step to converge. The incompressible

NSE problem, however, remains nonlinear in the staggered case and thus dominates the necessary amount of Newton iterations for the whole partitioned problem.

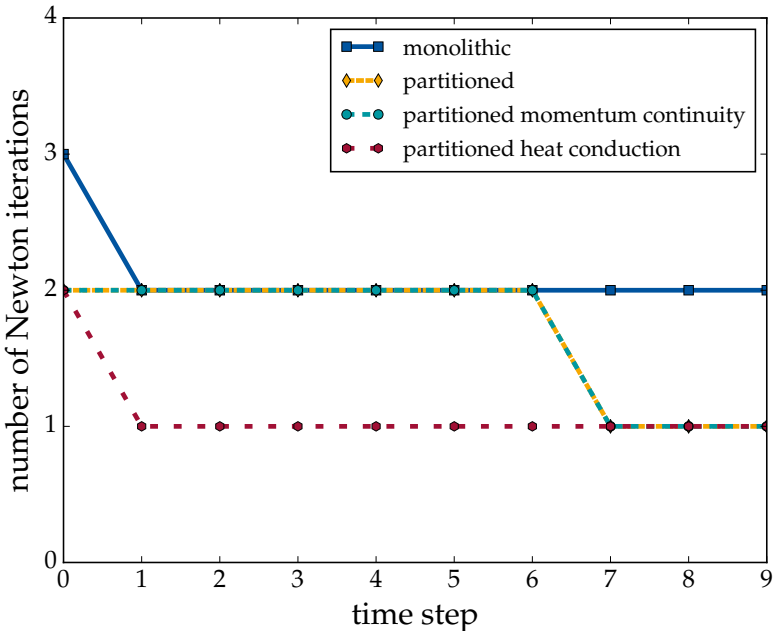


Figure 36: Number of Newton iterations needed to converge in each time step. For the partitioned approach the number of Newton iterations needed for the convergence of each subproblem are also provided.

It seems that solving two smaller problems requires fewer Newton iterations than solving the monolithic system. But these results have to be put into perspective regarding the very simple test problem. Here, the temperature has no influence on the velocity and pressure fields, which means that wrong or less accurate predictions of the temperature do not worsen the convergence of the NSE problem. This could be very different for bidirectional dependencies, e.g., if temperature dependent material parameters are considered.

The exact influence of the discussed approaches on the performance still has to be investigated. However, one can expect that the strongly partitioned approach will sometimes be faster, as solving two small linear systems might be more efficient than solving a large one.

8. Conclusion and Outlook

Within the first part of this thesis, the novel continuum mechanics simulation software CAMPIGA has been extended by new data types, which are able to determine the derivative of a computation together with its value. In the context of CAMPIGA, this functionality is applied to automatically determine the Jacobian required to solve a nonlinear equation system by Newton's method. This nonlinear equation system arises from the FEM discretization of a PDE system. The quantities of interest are the unknowns on the element nodes. This number can become very large for typical applications in science and engineering. Following the FEM approach, the Jacobian of the nonlinear residual is not determined globally, but on an element-level basis, where the number of unknowns is also much smaller. Similar to the residual, the global matrix is then assembled from the element-level contributions. The user can currently choose between an automatized matrix determination by AD or FD. However, it is still possible to provide a manual implementation of the residual's Jacobian for performance critical simulations. The CD capabilities have been integrated into the software by introducing a new wrapper for the frontend code. This new wrapper can be applied to any equation, it is also possible to assemble different matrix parts of the Jacobian by different means of CD. The correctness of the newly implemented features has been verified in two test cases. There were no visible differences between any of the implemented matrix determination methods, but their performance. The manual approach is always the fastest one, while a CD-based matrix assembly requires additional time. However, the overhead of automatized matrix assembly in the case of NSE becomes less significant, with increasing problem sizes. This is due to the fact that the total amount of time needed for the computation is dominated by the time required for solving the linear system. Regarding the convergence all methods behave similarly. The only caveat of the FD approach is the choice of the step width h, which can lead to oscillatory or even divergent behavior if chosen too small or too large. The AD approach shows exactly the same convergence as the manual approach.

The advantage of the AD approach is that it always provides an analytically exact solution for the derivative without the need, to first tune any problem-dependent parameters. However, it requires an overloaded implementation of every elemental function, which contains the analytical derivative. This is not required by the FD approach. Instead, it can be directly applied to any discretized weak formulation at the cost of a numerical error. The manual approach should be chosen whenever possible, especially, if more complex problems are considered, where the overall run-time is determined by the matrix assembly and not the time for the linear solver.

The second part has been dedicated to the solution of coupled problems, which arise in many fields of engineering applications. Besides a monolithic solution procedure, CAMPIGA is now able to solve volume coupled problems iteratively in a strongly cou-

pled fashion. In each Newton iteration two smaller linear systems are solved instead of one large system. Between the individual Newton iterations, the current solutions for the fields are exchanged among the two subproblems. The implementation of the coupling algorithm has been verified with the XNS simulation software. The investigation of a simple time dependent problem has shown, that the partitioned approach requires less Newton iterations in each time step than the monolithic approach. However, one has to keep in mind, that the chosen test case was very simple as the temperature field did not influence the velocity and pressure field. This means that inaccurate temperature predictions had no impact on the convergence of the incompressible Navier-Stokes system. However, being able to solve a couple of smaller systems instead of one large system, may be beneficial as soon as more realistic three dimensional engineering applications are considered.

As CAMPIGA is in a relatively early stage of the development process, there are still additional features which might be considered: The CD framework is currently only applied to the unknowns but one could extend it to data structures, which compute the gradients of the shape functions and the transformation from an arbitrary element to the reference element. This could be relevant for obtaining automatized derivatives of objective functions used in shape optimization problems. One could also try to improve the implemented CD methods. One option would be to try out a reverse mode implementation of AD or implement it using expression templates in order to reduce the associated overhead. In order to make the FD approach more viable, one could think about the implementation of an adaptive step width, to overcome the problem of selecting an appropriate step width manually.

Regarding coupling strategies, the code could be modified to solve an arbitrary number of problems using the partitioned approach. Additional work can be dedicated to extend the framework also to surface coupled problems, which would once again be useful for the investigation of fluid-structure interaction problems. The monolithic solution strategy can also be improved, e.g. by the implementation of special preconditioners, which account for different orders of magnitude in the involved fields. This could help to improve the condition of the monolithic system matrix. Another interesting aspect, which could be examined further, is a performance comparison of the discussed coupling approaches.

A. Derivatives of the Model Equations

This section contains the derivation for the derivatives of the model equations used in this thesis.

A.1. Mathematical Preliminaries

This section is supposed to clarify some of the mathematical notation, which will be used for the following derivations.

In analogy to the notation in [13], the divergence of a second order tensor $\mathbf{T} \in \mathbb{R}^{n\times n}$ will be computed row-wise, that is

$$[\boldsymbol{\nabla} \cdot \mathbf{T}]_i = \sum_{j=1}^{n} \frac{\partial T_{ij}}{\partial x_j} \,. \tag{A.1}$$

When deriving weak formulations, it is more convenient to write down the PDEs using the following conventions according to [9]: First, the summation sign will be dropped, whenever the bounds of the summation are clear. The summation will always be performed over recurring indices in the same expression. Using this notation, the divergence of Equation (A.1) can be written as follows:

$$\frac{\partial T_{ij}}{\partial x_j} \equiv \sum_{j=1}^{n} \frac{\partial T_{ij}}{\partial x_j} \,. \tag{A.2}$$

This notation can be further extended for partial derivatives. The component with respect to which the derivative is taken will be indicated in the subscript seperated by a colon. Thus Equation (A.2) becomes

$$T_{ij,j} \equiv \frac{\partial T_{ij}}{\partial x_j} \,. \tag{A.3}$$

A.2. Continuity Equation

The strong form of the continuity equation reads

$$v_{j,j} = 0 \,. \tag{A.4}$$

The unknown velocity field is discretized using the following ansatz:

$$v_i = \hat{v}_i^b \, \phi_v^b \tag{A.5}$$

The resulting weak formulation yields

$$C^A = \int_\Omega \hat{v}_j^b \, \phi_{v,j}^b \, \phi_{\mathsf{C}}^A \, \mathrm{d}\Omega \,, \tag{A.6}$$

where it was used, that

$$v_{i,j} = \hat{v}_i^b \; \phi_{v,j}^b \, . \tag{A.7}$$

We can now compute the derivative with respect to the unknown pressure, velocity and temperature values:

$$\frac{\partial C^A}{\partial \hat{p}^\alpha} = 0 \tag{A.8}$$

$$\frac{\partial C^A}{\partial \hat{v}_k^\beta} = \int_\Omega \phi_{v,k}^\beta \; \phi_{\mathsf{C}}^A \; \mathrm{d}\Omega \tag{A.9}$$

$$\frac{\partial C^A}{\partial \hat{T}^\gamma} = 0 \, , \tag{A.10}$$

or in vector notation

$$\frac{\partial C^A}{\partial \hat{p}^\alpha} = 0 \tag{A.11}$$

$$\frac{\partial C^A}{\partial \hat{\boldsymbol{v}}^\beta} = \int_\Omega \boldsymbol{\nabla} \phi_v^\beta \; \phi_{\mathsf{C}}^A \; \mathrm{d}\Omega \tag{A.12}$$

$$\frac{\partial C^A}{\partial \hat{T}^\gamma} = 0 \, . \tag{A.13}$$

A.3. Momentum Equation

The strong form of the momentum equation is

$$M_i^B = \underbrace{\int_\Omega \rho \; v_j \; v_{i,j} \; \phi_{\mathsf{M}}^B \; \mathrm{d}\Omega}_{M_{i,\mathsf{conv}}^B \text{, convective part}} + \underbrace{\int_\Omega \sigma_{ij} \; \phi_{\mathsf{M},j}^B \; \mathrm{d}\Omega}_{M_{i,\mathsf{visc}}^B \text{, viscous part}} \, . \tag{A.14}$$

For the material law, we have

$$\sigma_{ij} = \eta \; (v_{j,i} + v_{i,j}) - p \; \delta_{ij} \, . \tag{A.15}$$

For the pressure, the following ansatz is used

$$p = \hat{p}^a \; \phi_p^a \, . \tag{A.16}$$

The derivatives are computed for each part separately.

A.3.1. Convective Part

The convective part does neither depend on the pressure nor on the temperature:

$$\frac{\partial M_{i,\mathsf{conv}}^B}{\partial \hat{p}^\alpha} = 0 \tag{A.17}$$

$$\frac{\partial M_{i,\mathsf{conv}}^B}{\partial \hat{T}^\gamma} = 0 \, . \tag{A.18}$$

When deriving with respect to the velocity nodes, the following derivatives will be needed:

$$\frac{\partial v_i}{\partial \hat{v}_k^\beta} = \phi_v^\beta \, \delta_{ik} \tag{A.19}$$

$$\frac{\partial v_{i,j}}{\partial \hat{v}_k^\beta} = \phi_{v,j}^\beta \, \delta_{ik} \, . \tag{A.20}$$

With that, the derivative with respect to the velocity yields:

$$\frac{\partial M_{i,\mathsf{conv}}^B}{\partial \hat{v}_k^\beta} = \int_\Omega \rho \, \frac{\partial}{\partial \hat{v}_k^\beta} \left(v_j \, v_{i,j} \right) \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega \tag{A.21}$$

$$= \int_\Omega \rho \left(\frac{\partial v_j}{\partial \hat{v}_k^\beta} \, v_{i,j} + v_j \, \frac{\partial v_{i,j}}{\partial \hat{v}_k^\beta} \right) \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega \tag{A.22}$$

$$= \int_\Omega \rho \underbrace{\left(\phi_v^\beta \, \delta_{jk} \, v_{i,j} + v_j \, \phi_{v,j}^\beta \, \delta_{jk} \right)}_{T_{ik}^{\mathsf{conv}} :=} \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega \tag{A.23}$$

For the components of the tensor $\mathbf{T}^{\mathsf{conv}}$ it holds:

$$T_{ik}^{\mathsf{conv}} = \phi_v^\beta \, \delta_{jk} \, v_{i,j} + v_j \, \phi_{v,j}^\beta \, \delta_{jk} \tag{A.24}$$

$$= \phi_v^\beta \, v_{i,k} + \left(\boldsymbol{v} \cdot \boldsymbol{\nabla} \phi_v^\beta \right) \delta_{jk} \, , \tag{A.25}$$

or in tensor notation

$$\mathbf{T}^{\mathsf{conv}} = \phi_v^\beta \, \boldsymbol{\nabla} \boldsymbol{v} + \left(\boldsymbol{v} \cdot \boldsymbol{\nabla} \phi_v^\beta \right) \, \mathbf{Id} \, . \tag{A.26}$$

The derivative of the convective part with respect to the velocity components becomes

$$\frac{\partial \boldsymbol{M}_{\mathsf{conv}}^B}{\partial \hat{\boldsymbol{v}}^\beta} = \int_\Omega \rho \, \mathbf{T}^{\mathsf{conv}} \, \phi_{\mathsf{M}}^B \, \mathrm{d}\Omega \, . \tag{A.27}$$

A.3.2. Viscous Part

The following material derivatives come in handy when computing the derivatives of the momentum equation

$$\frac{\partial \sigma_{ij}}{\partial p} = -\delta_{ij} \tag{A.28}$$

$$\frac{\partial \sigma_{ij}}{\partial \hat{p}^\alpha} = \frac{\partial \sigma_{ij}}{\partial p} \, \frac{\partial p}{\partial \hat{p}^\alpha} \tag{A.29}$$

$$\frac{\partial \sigma_{ij}}{\partial v_{m,n}} = \eta \left(\frac{\partial v_{j,i}}{\partial v_{m,n}} + \frac{\partial v_{i,j}}{\partial v_{m,n}} \right) = \eta \left(\delta_{jm} \, \delta_{in} + \delta_{im} \, \delta_{jn} \right) \tag{A.30}$$

$$\frac{\partial \sigma_{ij}}{\partial \hat{v}_k^\beta} = \frac{\partial \sigma_{ij}}{\partial v_{m,n}} \, \frac{\partial v_{m,n}}{\partial \hat{v}_k^\beta} \, . \tag{A.31}$$

Additionally we need these derivatives of the ansatz functions

$$\frac{\partial p}{\partial \hat{p}^\alpha} = \phi_p^\alpha \tag{A.32}$$

$$\frac{\partial v_{i,j}}{\partial \hat{v}_k^\beta} = \phi_{v,j}^\beta \, \delta_{ik} \, . \tag{A.33}$$

For the temperature we have

$$\frac{\partial M_{i,\mathsf{visc}}^B}{\partial \hat{T}^\gamma} = 0 \, . \tag{A.34}$$

First, we compute the derivative with respect to the pressure unknowns

$$\frac{\partial M_{i,\mathsf{visc}}^B}{\partial \hat{p}^\alpha} = \int_\Omega \frac{\partial \sigma_{ij}}{\partial \hat{p}^\alpha} \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.35}$$

$$= \int_\Omega -\delta_{ij} \, \phi_p^\alpha \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.36}$$

$$= \int_\Omega -\phi_p^\alpha \, \phi_{\mathsf{M},i}^B \, \mathsf{d}\Omega \, , \tag{A.37}$$

or in vector notation

$$\frac{\partial \boldsymbol{M}_{\mathsf{visc}}^B}{\partial \hat{p}^\alpha} = - \int_\Omega \phi_p^\alpha \, \boldsymbol{\nabla} \phi_{\mathsf{M}}^B \, \mathsf{d}\Omega \, . \tag{A.38}$$

The derivative with respect to the velocity unknowns yields

$$\frac{\partial M_{i,\mathsf{visc}}^B}{\partial \hat{v}_k^\beta} = \int_\Omega \frac{\partial \sigma_{ij}}{\partial \hat{v}_k^\beta} \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.39}$$

$$= \int_\Omega \frac{\partial \sigma_{ij}}{\partial v_{m,n}} \, \phi_{v,n}^\beta \, \delta_{mk} \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.40}$$

$$= \int_\Omega \frac{\partial \sigma_{ij}}{\partial v_{k,n}} \, \phi_{v,n}^\beta \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.41}$$

$$= \int_\Omega \eta \left(\delta_{jk} \, \delta_{in} + \delta_{ik} \, \delta_{jn} \right) \, \phi_{v,n}^\beta \, \phi_{\mathsf{M},j}^B \, \mathsf{d}\Omega \tag{A.42}$$

$$= \int_\Omega \eta \underbrace{\left(\phi_{\mathsf{M},k}^B \, \phi_{v,i}^\beta + \delta_{ik} \, \phi_{\mathsf{M},j}^B \, \phi_{v,j}^\beta \right)}_{T_{ik}^{\mathsf{visc}} :=} \mathsf{d}\Omega \, . \tag{A.43}$$

The components of the tensor $\mathbf{T}^{\mathsf{visc}}$ can be written as

$$T_{ik}^{\mathsf{visc}} = \phi_{\mathsf{M},k}^B \, \phi_{v,i}^\beta + \delta_{ik} \, \phi_{\mathsf{M},j}^B \, \phi_{v,j}^\beta \tag{A.44}$$

$$= \phi_{\mathsf{M},k}^B \, \phi_{v,i}^\beta + \left(\boldsymbol{\nabla} \phi_{\mathsf{M}}^B \cdot \boldsymbol{\nabla} \phi_v^\beta \right) \delta_{ik} \, , \tag{A.45}$$

or in vector notation

$$\mathbf{T}^{\mathsf{visc}} = \boldsymbol{\nabla} \phi_{\mathsf{M}}^B \otimes \boldsymbol{\nabla} \phi_v^\beta + \left(\boldsymbol{\nabla} \phi_{\mathsf{M}}^B \cdot \boldsymbol{\nabla} \phi_v^\beta \right) \, \mathbf{Id} \, . \tag{A.46}$$

The derivative of the viscous part with respect to the velocity components becomes

$$\frac{\partial \boldsymbol{M}_{\mathsf{visc}}^B}{\partial \hat{\boldsymbol{v}}^\beta} = \int_\Omega \eta \, \mathbf{T}^{\mathsf{visc}} \, \mathsf{d}\Omega \, . \tag{A.47}$$

A.4. Heat Conduction Equation

The strong form of the heat conduction equation reads:

$$H^C = \underbrace{\int_\Omega \rho c_p \, v_j \, T_{,j} \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega}_{H^C_{\mathsf{conv}},\ \text{convective part}} + \underbrace{\int_\Omega \kappa \, T_{,j} \, \phi^C_{\mathsf{H},j} \, \mathrm{d}\Omega}_{H^C_{\mathsf{diff}},\ \text{diffusive part}} . \tag{A.48}$$

For the temperature, the following ansatz is used:

$$T = \hat{T}^c \, \phi^c_T \, . \tag{A.49}$$

A.4.1. Convective Part

The convective part of the heat conduction equation does not depend on the pressure, thus

$$\frac{\partial H^C_{\mathsf{conv}}}{\partial \hat{p}^\alpha} = 0 \, . \tag{A.50}$$

For the derivative with respect to the velocity we obtain

$$\frac{\partial H^C_{\mathsf{conv}}}{\partial \hat{v}^\beta_k} = \int_\Omega \rho c_p \, \delta_{jk} \, \phi^\beta_v \, T_{,j} \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega \tag{A.51}$$

$$= \int_\Omega \rho c_p \, \phi^\beta_v \, T_{,k} \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega \, , \tag{A.52}$$

or in vector notation

$$\frac{\partial H^C_{\mathsf{conv}}}{\partial \hat{\boldsymbol{v}}^\beta} = \int_\Omega \rho c_p \, \phi^\beta_v \, \boldsymbol{\nabla} T \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega \, . \tag{A.53}$$

The derivative with respect to the temperature yields

$$\frac{\partial H^C_{\mathsf{conv}}}{\partial \hat{T}^\gamma} = \int_\Omega \rho c_p \, v_j \, \phi^c_{T,j} \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega \, , \tag{A.54}$$

where we can rewrite the sum inside the integral as dot product

$$\frac{\partial H^C_{\mathsf{conv}}}{\partial \hat{T}^\gamma} = \int_\Omega \rho c_p \, (\boldsymbol{v} \cdot \boldsymbol{\nabla} \phi^c_T) \, \phi^C_{\mathsf{H}} \, \mathrm{d}\Omega \, . \tag{A.55}$$

A.4.2. Diffusive Part

The diffusive part depends only on the temperature. Consequently it holds

$$\frac{\partial H^C_{\mathsf{diff}}}{\partial \hat{p}^\alpha} = 0 \tag{A.56}$$

$$\frac{\partial H^C_{\mathsf{diff}}}{\partial \hat{v}^\beta_i} = 0 \, . \tag{A.57}$$

For the derivative with respect to the temperature we get

$$\frac{\partial H^C_{\mathsf{diff}}}{\partial \hat{T}^\gamma} = \int_\Omega \kappa \, \phi^c_{T,j} \, \phi^C_{\mathsf{H},j} \, \mathrm{d}\Omega \,, \tag{A.58}$$

or in vector notation

$$\frac{\partial H^C_{\mathsf{diff}}}{\partial \hat{T}^\gamma} = \int_\Omega \kappa \, (\boldsymbol{\nabla} \phi^c_T \cdot \boldsymbol{\nabla} \phi^C_{\mathsf{H}}) \, \mathrm{d}\Omega \,. \tag{A.59}$$

B. Profiling Data

This section contains the data for the plots related to the performance analysis presented in Section 4.1.

B.1. Percentage of the total runtime

Method	Assembly	Solution	Other
Manual	26.1 %	71.4 %	2.5 %
AD	61.0 %	37.9 %	1.1 %
FD	56.7 %	41.5 %	1.8 %

Table B.1: Without compiler optimization, $n_{\text{el}} = 100$.

Method	Assembly	Solution	Other
Manual	14.1 %	83.0 %	2.9 %
AD	34.8 %	62.9 %	2.3 %
FD	33.5 %	64.3 %	2.2 %

Table B.2: Without compiler optimization, $n_{\text{el}} = 400$.

Method	Assembly	Solution	Other
Manual	15.1 %	80.4 %	4.5 %
AD	43.4 %	53.7 %	2.9 %
FD	49.0 %	49.4 %	1.6 %

Table B.3: With compiler optimization, $n_{\text{el}} = 100$.

Method	Assembly	Solution	Other
Manual	6.5 %	89.5 %	4.0 %
AD	22.5 %	74.2 %	3.3 %
FD	24.1 %	73.0 %	2.9 %

Table B.4: With compiler optimization, $n_{\text{el}} = 400$.

B.2. Total runtime

Method	Total	Assembly	Solution
Manual	0.30 s	78.30 ms	214.20 ms
AD	0.48 s	292.80 ms	181.92 ms
FD	0.50 s	283.50 ms	207.50 ms

Table B.5: Without compiler optimization, $n_{\text{el}} = 100$.

Method	Total	Assembly	Solution
Manual	2.39 s	336.99 ms	1983.70 ms
AD	2.96 s	1030.08 ms	1861.84 ms
FD	3.18 s	1065.30 ms	2044.74 ms

Table B.6: Without compiler optimization, $n_{\text{el}} = 400$.

Method	Total	Assembly	Solution
Manual	0.19 s	28.69 ms	152.76 ms
AD	0.29 s	125.86 ms	155.73 ms
FD	0.30 s	147.00 ms	148.20 ms

Table B.7: With compiler optimization, $n_{\text{el}} = 100$.

Method	Total	Assembly	Solution
Manual	1.70 s	110.50 ms	1521.50 ms
AD	2.03 s	456.75 ms	1506.26 ms
FD	2.09 s	503.69 ms	1525.70 ms

Table B.8: With compiler optimization, $n_{\text{el}} = 400$.

References

[1] *GCC 9.2 Manual*. URL https://gcc.gnu.org/onlinedocs/gcc-9.2.0/gcc.pdf.

[2] Pierre Aubert, Nicolas Di Césaré, and Olivier Pironneau. Automatic differentiation in C++ using expression templates and application to a flow control problem. *Computing and Visualization in Science*, 3(4):197–208, Jan 2001. ISSN 1432-9360. doi: 10.1007/s007910000048. URL https://doi.org/10.1007/s00791000004.

[3] Roscoe A. Bartlett, David M. Gay, and Eric T. Phipps. Automatic differentiation of C++ codes for large-scale scientific computing. In *Computational Science – ICCS 2006*, pages 525–532. Springer Berlin Heidelberg, 2006. doi: 10.1007/11758549_73. URL https://link.springer.com/content/pdf/10.1007%2F11758549_73.pdf.

[4] C. Bendtsen and Ole Stauning. FADBAD, a flexible C++ package for automatic differentiation. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark, aug 1996. URL http://www2.imm.dtu.dk/~kajm/FADBAD/tr17_96.pdf.

[5] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR – generating derivative codes from fortran programs. *Scientific Programming*, 1(1):11–29, 1992. doi: 10.1155/1992/717832. URL http://downloads.hindawi.com/journals/sp/1992/717832.pdf.

[6] Christian H. Bischof, Gordon D. Pusch, and Ralf Knoesel. Sensitivity analysis of the MM5 weather model using automatic differentiation. *Computers in Physics*, 10(6):605, 1996. doi: 10.1063/1.168585. URL https://www.researchgate.net/publication/2786042_Sensitivity_Analysis_of_the_MM5_Weather_Model_Using_Automatic_Differentiation.

[7] J. Blasco, R. Codina, and A. Huerta. A fractional step method for the incompressible Navier-Stokes equations related to a predictor-multicorrector algorithm. *International Journal for Numerical Methods in Fluids*, 28(10):1391–1419, 1998. ISSN 1097-0363. doi: 10.1002/(SICI)1097-0363(19981230)28:10<1391::AID-FLD699>3.0.CO;2-5. URL https://onlinelibrary.wiley.com/doi/epdf/10.1002/%28SICI%291097-0363%2819981230%2928%3A10%3C1391%3A%3AAID-FLD699%3E3.0.CO%3B2-5.

[8] E. Borgonovo, S. Gatti, and L. Peccati. What drives value creation in investment projects? An application of sensitivity analysis to project finance transactions. *European Journal of Operational Research*, 205(1):227–236, aug 2010. doi: 10.1016/j.ejor.2009.12.006. URL http://www.centrobaffi.unibocconi.it/wps/allegatiCTP/what%20drivesgatti.pdf.

[9] Esben Byskov. *Index Notation, the Summation Convention, and a Little About Tensor Analysis*, pages 511–519. Springer Netherlands, Dordrecht, 2013. ISBN 978-94-007-5766-0. doi: 10.1007/978-94-007-5766-0_31. URL https://doi.org/10.1007/978-94-007-5766-0_31.

[10] V. M. Calo, N. F. Brasher, Y. Bazilevs, and T. J. R. Hughes. Multiphysics model for blood flow and drug transport with application to patient-specific coronary artery flow. *Computational Mechanics*, 43(1):161–177, aug 2008. doi: 10.1007/s00466-008-0321-z. URL https://link.springer.com/article/10.1007/s00466-008-0321-z.

[11] Rama Cont, Romain Deguest, and Giacomo Scandolo. Robustness and sensitivity analysis of risk measurement procedures. *Quantitative Finance*, 10(6): 593 – 606, Jun 2010. doi: 10.1080/14697681003685597. URL https://hal.archives-ouvertes.fr/hal-00413729/file/robustriskarxiv.pdf.

[12] Wolfgang Dahmen and Arnold Reusken. *Numerik für Ingenieure und Naturwissenschaftler*. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-76493-9. URL https://link.springer.com/book/10.1007/978-3-540-76493-9.

[13] Jean Donea and Antonio Huerta. *Finite Element Methods for Flow Problems*. John Wiley & Sons, Ltd, apr 2003. doi: 10.1002/0470013826. URL https://onlinelibrary.wiley.com/doi/book/10.1002/0470013826.

[14] David Eberly. Derivative approximation by finite differences. 2001. URL https://www.geometrictools.com/Documentation/FiniteDifferences.pdf.

[15] Carlos A. Felippa, K.C. Park, and Charbel Farhat. Partitioned analysis of coupled mechanical systems. *Computer Methods in Applied Mechanics and Engineering*, 190 (24):3247 – 3270, March 2001. ISSN 0045-7825. doi: https://doi.org/10.1016/S0045-7825(00)00391-1. URL http://www.sciencedirect.com/science/article/pii/S0045782500003911. Advances in Computational Methods for Fluid-Structure Interaction.

[16] Christiane Förster, Wolfgang A. Wall, and Ekkehard Ramm. Artificial added mass instabilities in sequential staggered coupling of nonlinear structures and incompressible viscous flows. *Computer Methods in Applied Mechanics and Engineering*, 196(7):1278–1293, jan 2007. doi: 10.1016/j.cma.2006.09.002. URL https://www.sciencedirect.com/science/article/pii/S0045782506002544.

[17] Peter Gottschling. *Forschung mit modernem C++: C++17-Intensivkurs für Wissenschaftler, Ingenieure und Programmierer*. Carl Hanser Verlag München, 2019. URL https://www.hanser-elibrary.com/isbn/9783446458468.

[18] Andreas Griewank and Andrea Walther. *Evaluating Derivatives*. Society for Industrial and Applied Mathematics, Jan 2008. doi: 10.1137/1.9780898717761. URL https://epubs.siam.org/doi/book/10.1137/1.9780898717761.

[19] Andreas Griewank, David Juedes, and Jean Utke. Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 22(2):131–167, jun 1996. doi: 10.1145/229473.229474. URL https://dl.acm.org/doi/10.1145/229473.229474.

[20] Laurent Hascoet and Valérie Pascual. The tapenade automatic differentiation tool. *ACM Transactions on Mathematical Software*, 39(3):1–43, apr 2013. doi: 10.1145/2450153.2450158. URL https://dl.acm.org/doi/10.1145/2450153.2450158.

[21] Robin J. Hogan. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software*, 40(4):1–16, Jul 2014. doi: 10.1145/2560359. URL http://www.met.reading.ac.uk/~swrhgnrj/publications/adept.pdf.

[22] David W. Juedes. A taxonomy of automatic differentiation tools. 1992. URL https://pdfs.semanticscholar.org/3faa/d0d6d7bb21c53d791d661a1f2009604d97b8.pdf.

[23] Drosos Kourounis, Leonidas Gergidis, Michael Saunders, Andrea Walther, and Olaf Schenk. Compile-time symbolic differentiation using C++ expression templates. 2017. URL https://arxiv.org/pdf/1705.01729.pdf.

[24] Jeffrey Larson, Matt Menickelly, and Stefan M. Wild. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, may 2019. doi: 10.1017/s0962492919000060. URL http://www.optimization-online.org/DB_FILE/2019/04/7153.pdf.

[25] Tim Lieuwen. Modeling premixed combustion-acoustic wave interactions: A review. *Journal of Propulsion and Power - J PROPUL POWER*, 19, 09 2003. doi: 10.2514/2.6193. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.568.4994&rep=rep1&type=pdf.

[26] Charles C. Margossian. A review of automatic differentiation and its efficient implementation. 2018. doi: 10.1002/WIDM.1305. URL https://arxiv.org/pdf/1811.05031.pdf.

[27] Bernd Markert. *Weak or Strong: On Coupled Problems in Continuum Mechanics*. Habilitation thesis, Institute of Applied Mechanics (Civil Engineering), University of Stuttgart, Germany, 2010. URL https://elib.uni-stuttgart.de/handle/11682/359.

[28] Toshiyuki Nakata and Hao Liu. A fluid-structure interaction model of insect flight with flexible wings. *Journal of Computational Physics*, 231(4):1822 – 1847, 2012. ISSN 0021-9991. doi: https://doi.org/10.1016/j.jcp.2011.11.005. URL http://www.sciencedirect.com/science/article/pii/S0021999111006474.

[29] Sri Hari Krishna Narayanan, Boyana Norris, and Beata Winnicka. ADIC2: Development of a component source transformation system for differentiating C and C++. *Procedia Computer Science*, 1(1):1845–1853, may 2010. doi: 10.1016/j.procs.2010.04.206. URL https://www.sciencedirect.com/science/article/pii/S1877050910002073?via%3Dihub.

[30] Uwe Naumann. *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, Jan 2011. doi: 10.1137/1.9781611972078. URL https://epubs.siam.org/doi/book/10.1137/1.9781611972078.

[31] Jorge Nocedal and Stephen J. Wright, editors. *Numerical Optimization*. Springer-Verlag, 1999. doi: 10.1007/b98874. URL http://www.apmath.spbu.ru/cnsa/pdf/monograf/Numerical_Optimization2006.pdf.

[32] J.Blair Perot. An analysis of the fractional step method. *Journal of Computational Physics*, 108(1):51–58, sep 1993. doi: 10.1006/jcph.1993.1162. URL https://www.sciencedirect.com/science/article/pii/S0021999183711629.

[33] Eric Phipps and Roger Pawlowski. Efficient expression templates for operator overloading-based automatic differentiation. 2012. URL https://arxiv.org/pdf/1205.3506.pdf.

[34] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, mar 1960. doi: 10.1093/comjnl/3.3.175. URL https://academic.oup.com/comjnl/article/3/3/175/345501.

[35] Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes. Algorithmic differentiation of a complex C++ code with underlying libraries. *Procedia Computer Science*, 18:208–217, 2013. doi: 10.1016/j.procs.2013.05.184. URL https://www.sciencedirect.com/science/article/pii/S187705091300327X.

[36] Michael Schlottke-Lakemper, Matthias Meinke, and Wolfgang Schröder. A hybrid discontinuous galerkin-finite volume method for computational aeroacoustics. In Andreas Dillmann, Gerd Heller, Ewald Krämer, Claus Wagner, and Christian Breitsamter, editors, *New Results in Numerical and Experimental Fluid Mechanics X*, pages 743–753, Cham, 2016. Springer International Publishing. ISBN 978-3-319-27279-5. URL http://www.aia.rwth-aachen.de/vlueb/staff/homes/82/publications/paper.pdf.

[37] Wolfgang Schröder. *Fluidmechanik*. Verlagsgruppe Mainz, 2014. ISBN 978-3-95886-221-0.

[38] Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch. OpenAD/F: A modular open-source tool for automatic differentiation of fortran codes. *ACM Transactions on Mathematical Software*, 34(4):1–36, jul 2008. doi: 10.1145/1377596.1377598. URL https://dl.acm.org/doi/10.1145/1377596.1377598.

[39] Francesc Verdugo and Wolfgang A. Wall. Unified computational framework for the efficient solution of n-field coupled problems with monolithic schemes. *Computer Methods in Applied Mechanics and Engineering*, 310:335–366, oct 2016. doi: 10.1016/j.cma.2016.07.016. URL https://www.sciencedirect.com/science/article/pii/S0045782516307575.

[40] Florian vom Lehn, Liming Cai, and Heinz Pitsch. Sensitivity analysis, uncertainty quantification, and optimization for thermochemical properties in chemical kinetic combustion models. *Proceedings of the Combustion Institute*, 37(1): 771 – 779, 2019. ISSN 1540-7489. doi: https://doi.org/10.1016/j.proci.2018.06.188. URL http://www.sciencedirect.com/science/article/pii/S1540748918303742.

[41] P. Zunino, C. D'Angelo, L. Petrini, C. Vergara, C. Capelli, and F. Migliavacca. Numerical simulation of drug eluting coronary stents: Mechanics, fluid dynamics and drug release. *Computer Methods in Applied Mechanics and Engineering*, 198(45-46):3633–3644, sep 2009. doi: 10.1016/j.cma.2008.07.019. URL https://www.sciencedirect.com/science/article/pii/S0045782508002703.